UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

A project report submitted for the continuation towards a
MEng Computer Science

Supervisor: Prof David Millard
Examiner: Prof Danesh Tarapore

Prototype of an Automated LiDAR-to-Mesh Pipeline Implemented in Unity: A Winchester Case Study

by Aleksander Pilski

December 16, 2025

Abstract

This project developed a prototype of an automated pipeline to convert airborne LiDAR data into editable 3D mesh models within Unity, using Winchester (UK) as a test case. The pipeline imported LAZ point clouds and DTM rasters, preserved ground and building elevations, classified points into ground, roof, and vegetation categories, and generated lightweight, editable Unity prefabs for first-person exploration. Processing remained in the point cloud domain for as long as possible before meshing to maximise control over feature extraction. Performance on a $1 \times 1\,\text{km}$ tile proved acceptable, with key stages completing within minutes, although full tile meshing remained slow on standard hardware.

The project confirmed that real-world data can significantly accelerate realistic urban environment creation in game engines, though challenges such as memory limitations, mesh anomalies, and classification errors persisted. Key lessons included the need to externalise heavy data preparation outside Unity, the importance of a modular workflow, and the future benefit of high-performance computing resources. Although developed individually and constrained by hardware limits, the prototype provided a solid foundation for future improvements in scalability, visual fidelity, and automated quality assurance. Longer-term ambitions include refining building separation, enhancing classification with computer vision techniques, and ultimately creating a playable, interactive reconstruction of Winchester.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This project was my attempt to explore how real-world data could be used to ease the creation of realistic urban environments in games, aiming to reduce the time and manual effort usually required. Rather than following a strict development model, the process evolved through exploration, trial and error, and gradual refinement. The outcome was not a complete solution, but a working foundation that demonstrated the potential of this approach and offered a starting point for further development.

Several years ago, I became involved in the *Mount & Blade* modding community. One of the main challenges was creating large, persistent maps for the *Persistent World* mod, where players could form their own kingdoms around castles and interact in a shared environment. These maps were far larger than standard battle maps and required meticulous work with a developer-provided editor containing thousands of objects. Although some complete meshes were included, mapmakers had to invest countless hours adjusting terrain, placing objects, and testing collisions to produce functional and immersive maps. Some projects even recreated real or fictional locations with impressive detail.

While the time investment was significant, the results were rewarding — both visually and through community recognition, with top maps selected for server rotation. It was during this time that I first imagined automating parts of the process, particularly using real-world data to recreate locations. However, lacking the necessary skills, I shelved the idea as a long-term ambition.

Over the following years, I explored various games with different approaches to map realism, always keeping that idea in mind. Broadly, these approaches can be divided into **handcrafted realism** and **procedural generation**.

On the handcrafted side, games like *Grand Theft Auto V*, *Deus Ex: Mankind Divided*, and *Kingdom Come: Deliverance* reflect years of work from large teams, supported by substantial budgets and research. These titles achieved impressive realism through meticulous manual design, often involving field studies or historical research. *Cyberpunk 2077* also exemplified this approach, with urban planners contributing to the design of Night City.

In contrast, games like *Minecraft*, *No Man's Sky*, and *City of Brass* demonstrate the strengths of procedural generation. They offer vast or varied environments through algorithmic systems, enabling massive scale and diversity at the cost of fine-grained realism. While powerful, these systems often struggle to replicate the realistic feel of actual cities.

Modern game engines such as CryEngine and Unreal Engine have largely solved the problem of natural terrain generation. However, it remains unclear to what extent these or other tools have leveraged real-world data to streamline the creation of urban spaces, as proprietary methods and trade secrets often obscure the details. Independent projects, particularly in modding communities, increasingly use engines like Unity and Unreal Engine to build flexible editors — a method echoing my early experiences with *Mount & Blade*.

As part of my studies, I recently developed a game project using the Unity engine and deepened my understanding of Computer Vision and Database technologies. This combination of skills and interests presented the right moment to revisit the problem of creating realistic urban environments more efficiently.

Chapter 2

Literature Review

2.1 Literature Review

2.1.1 Why I Carried Out This Review

Before building the prototype, I needed to understand how others had turned raw geospatial data into models that a game engine can use. I therefore asked three practical questions:

(i) Which data sources give the most useful information?

(ii) Which processing steps separate ground, buildings, and vegetation with reasonable computing effort?

(iii) Which methods could I transfer into Unity within my limited time and skills?

2.1.2 Review and Discussion

2.1.2.1 DTM Filtering and Interpolation

Kraus and Pfeifer [17] combined terrain filtering and interpolation in one routine that worked even on steep, wooded slopes. Their method relied on a hierarchical robust-interpolation scheme: coarse layers detected the main terrain trend, while finer layers removed local outliers caused by trees or buildings. They evaluated the approach on Alpine test sites and reported mean terrain errors below one

half of the point spacing, establishing a clear performance target for later filters. Abdeldayem's Automatic Weighted Splines Filter (AWSF) [1] refined this idea by adding a spatial weighting function to cubic smoothing splines. AWSF was compared with fourteen established filters on mountainous forest plots drawn from the ISPRS benchmark and consistently ranked in the top three for both root-mean-square error and type-II error (missed ground points). These results encouraged me to keep spline-based filters in mind if higher accuracy became necessary. Matwij et al. [22] transferred similar filtering principles to structural monitoring: multi-epoch TLS scans of bridge pillars were aligned, filtered, and differenced to highlight millimetre-level deformations. Although this application lay outside the present project, it demonstrated that careful ground extraction underpins reliable change detection.

2.1.2.2 Building Reconstruction from Airborne and Terrestrial Laser Scanning

Elberink and Vosselman [9] matched point-cloud graphs to a library of target building graphs. Each node represented a planar segment and each edge a boundary line; matching quality scores revealed missing or extra parts. Their tests on Dutch suburban blocks showed that 85 Dorninger and Pfeifer [5] presented a fully automatic pipeline: region growing detected planar patches, straight-skeleton tracing produced building outlines, and model primitives were assembled into CityGML output. They processed an entire Austrian town (around 2 500 buildings) in under two hours on a single workstation, showing that full automation was already feasible. Rottensteiner and Briese [25] fused DTM and digital-surface models to isolate off-terrain points before applying curvature-based segmentation; the authors stressed that a coarse DTM was sufficient, underlining the value of a point-level approach. Further refinements focused on roofs. Jochem et al. [16] extracted planar roof facets and estimated incident solar radiation using hourly meteorological data; they reported an average annual error of 6 Wang et al. [31] introduced multi-directional band projection to detect roof edges: points were projected onto a series of azimuth slices, creating one-dimensional edge signatures that were then merged. The algorithm required only three numerical parameters and processed two square kilometres of LiDAR in under five minutes, demonstrating both speed and stability. Feng et al. [10] proposed an improved minimum-bounding-rectangle (IMBR) algorithm that iteratively refined candidate rectangles by density and angle criteria, achieving footprint completeness above 92 Pu and Vosselman [24] combined TLS with oblique images to improve façade detail: strong image lines

replaced noisy laser edges, and textures were draped automatically, resulting in photorealistic street models with sub-pixel reprojection error.

2.1.2.3 Vectorisation and Footprint Modelling from LiDAR and Imagery

Li and Xin [18] chained UNet segmentation, Faster-R-CNN bounding boxes, and a corner detector to convert 15 cm imagery into polygon footprints. A Delaunay triangulation stage enforced straight edges and right angles, reducing the average vertex error from 1.2 m to 0.3 m compared with raster masks. Van den Broeck and Goedemé [30] applied a multitask fully convolutional network to 7 cm orthophotos, predicting roof-part regions and boundaries in a single forward pass. Post-processing merged the two outputs into watertight roof-part polygons that fitted within 0.5 m of airborne LiDAR references for 93 Campoverde et al. [4] added an attention-based network that highlighted edge and corner cues, then assembled planar graphs through a greedy merging routine. Their cross-city evaluation on Berlin, Paris, and Chicago showed that the model generalised well to different roof styles and shadow conditions. Feng [10] and Wang [31] demonstrated that LiDAR-only footprints can reach similar accuracy if point projection or geometric constraints are used. Both methods produced regularised outlines whose maximum deviation from cadastral truth did not exceed two point spacings, meeting the tolerance for Level-of-Detail-2 city models.

2.1.2.4 Deep-Learning Segmentation and Classification

Xia et al. [34] combined dense dilated convolutions with a dual edge-region branch in DDLNet. Full-scale skip connections preserved fine detail, and an edge-guided loss improved boundary precision by 4 percentage points over baseline UNet on the Inria Aerial Image Labeling dataset. Wang et al. [33] introduced B-FGC-Net, which used a feature-highlight module, global-context pooling, and cross-level fusion. The model raised mean Intersection-over-Union from 70 Liu et al. [20] presented SeCondPoint, a semantics-conditioned framework that learned a joint feature–language space. By generating pseudo features aligned with text prompts, the network achieved genuine zero-shot segmentation: unseen object classes were identified with F1 scores above 50 Hui et al. [15] changed the unit of analysis from points to objects by enforcing multi-constraint graph grouping, recovering missing roof parts that normal-based methods overlooked. Shu et al. [27] designed

PGR-Net for interactive editing: user clicks generated distance maps that guided a recurrent refinement loop, reducing the number of corrections required by 30 Zhou et al. [37] targeted sparse heritage scans with MLGTM, a transformer that aggregated multi-scale geometric tokens; it improved class accuracy on the Terracotta Warriors dataset from 62

2.1.2.5 Mesh and Volume Generation

Gonizzi Barsanti et al. [13] reviewed fourteen meshing and voxelisation algorithms on five heritage case studies, including a Gothic cathedral and a Roman theatre. They compared geometric error, watertightness, and element quality against finite-element analysis (FEA) requirements. Algorithms that excelled on clean engineering data showed severe artefacts—such as self-intersections and thin slivers—when confronted with irregular, partially denoised scans. The authors concluded that geometric simplification and global volume consistency remain unsolved for complex monuments. Meng et al. [23] addressed volume estimation for open-pit mines by slicing UAV point clouds at 0.5 m intervals, applying Euclidean clustering within each slice, and building a concave hull for every cluster. Aggregating slice volumes reduced mean absolute error to 2.1 Shin et al. [26] and Spick [29] explored automated terrain texturing: heightmaps were split into slope and biome zones, then matched with texture atlases. Although texture quality was subjective, both studies reported frame rates above 60 fps on mid-range GPUs, demonstrating real-time feasibility. Yang et al. [35] proposed BI-Net, which blended autoencoder compression with GAN up-sampling in a bidirectional training loop. The network produced uniformly distributed synthetic point clouds whose nearest-neighbour spacing deviated by only 5

2.1.2.6 Visualisation in Game Engines

Wang et al. [32] created a virtual city in Unity3D by importing CAD blocks and 30 m DEM tiles into separate layers. Interactive attribute queries were driven by a MySQL back-end, showing that GIS databases could be linked to a commercial engine without bespoke middleware. Mat et al. [21] compared Unity, Cesium, and Landserf using ten terrain datasets ranging from 5 m to 90 m resolution. Unity offered the smoothest navigation at high resolutions, but Cesium handled large extents better through tiled streaming. Lian et al. [19] reconstructed indoor scenes with neural radiance fields, converted the implicit representation to 3D

Gaussian splats, and rendered them in Unity for immersive fire-fighting training. The system maintained 40 fps in an HMD and supported multi-user sessions via Photon Networking. Bonczak and Kontokosta [3] voxelised LiDAR scans of 1.1 million New York buildings into 1 m cubes to compute volume, surface area, and compactness indices. Processing ran on a 640-core cluster, and the results were stored in a PostgreSQL/PostGIS database, from which Unity could stream selected tiles on demand.

2.1.2.7 Other Relevant Work

Soriano-Cuesta et al. [28] visualised borehole logs in a 3D GIS by extruding stratigraphic columns and linking them to predictive ground models, a technique that could extend city models below ground. An [2] linked airborne LiDAR with sky-view factor analysis to map urban heat exposure; building height and tree canopies were both derived from the same point cloud, demonstrating multi-purpose use of a single survey. Gallerani et al. [11] combined LiDAR-based habitat models with population-viability analysis for a critically endangered bird, illustrating how volumetric metrics can inform conservation. Stroner et al. [38] trained Gaussian-mixture and neural classifiers on RGB values alone, achieving 85 Zhang et al. [36] introduced a parallel spatial data conversion engine (PSCE) that transformed one terabyte of geodata from LAS to a tiled octree in 27 minutes on 32 cores, pointing to a practical way to prepare large datasets for streaming.

2.1.3 What I Learned

Five clear lessons guided the prototype:

(1) **Use airborne LiDAR as the main source.** It records ground, roofs, and vegetation in one survey [17].

(2) **Work in the point cloud for as long as possible.** Detailed control is only available before converting to rasters [5, 1].

(3) **Add a simple classification step.** Even a basic ground/roof classifier improves later processing [20].

(4) **Write a small Unity plug-in rather than adopting large GIS systems.** Large voxel or NeRF pipelines were beyond my resources [3].

(5) **Keep future improvements in view.** Fast format conversion [36] and advanced texturing [19] remain priorities for later versions.

These points formed the basis for the tasks and risk assessment described next in Chapter 3.

Chapter 3

Project Management and Planning

This chapter explains how the ideas set out in Chapter 1 and Chapter 2 were turned into a workable plan. It covers

(1) the development approach I followed;

(2) the problem, requirements, scope, and goals;

(3) the way the work was scheduled and the main risks; and

(4) how testing, reflection, and evaluation guided day-to-day work.

3.1 Development approach

Because I worked alone, no single life-cycle model fitted. Progress ran through three overlapping phases:

Exploratory - Short Unity trials revealed basic limits, such as how height-map import softened vertical walls.

Incremental - Each success was strengthened and reused, reducing risk in small steps instead of betting on a big design up front.

Rapid prototypes - When a question blocked progress, I wrote a quick throw-away tool—a voxel viewer, a mesh slicer, a texture probe—to answer it, then moved on.

3.2 Problem, requirements, and scope

Problem

Building believable game cities normally calls for custom pipelines, large teams, and strong hardware. My aim was to reach a modest but convincing result from public data on a home computer.

Requirements

The pipeline had to meet four clear requirements:

(R1) produce a first result within hours, not weeks;

(R2) keep individual building models in their correct positions;

(R3) load directly into Unity as editable objects; and

(R4) allow first-person exploration at street level.

Scope

Work stayed inside one test tile (SU4829, central Winchester, UK) and used only data supplied by the Environment Agency through the EDINA Digimap service:

- LiDAR point cloud (average horizontal accuracy 25 cm);
- LiDAR DTM raster (50 cm grid);
- a 25 cm aerial photograph; and
- a 1 m topographic texture.

Road detail, rich texturing, and gameplay were left for future work.

Project goals

Lessons from the literature review became five goals:

(G1) Use the LiDAR point cloud for buildings and the 50 cm DTM for terrain.

(G2) Keep processing in the point cloud until meshing is required.

(G3) Insert a simple ground/roof/vegetation classifier early.

(G4) Deliver results through a compact Unity plug-in, not a full GIS stack.

(G5) Leave hooks for later upgrades such as faster converters or advanced texturing.

3.3 Planning and time use

I created a detailed Gantt chart (Appendix **??**) during the proposal stage, which I used as a general guide throughout the project. However, as the work progressed, the project naturally evolved from a problem-driven to a solution-driven process, structured more like a growing pipeline. This meant I could not always follow the original timeline strictly: each problem needed to be fully solved before I could move on to the next stage, leading to a more adaptive, sequential approach than I had initially planned.

3.4 Risk management

Three risks required active intervention during the project:

- **Memory spikes in Unity** — this issue was not predicted, even with my previous experience using Unity. It arose when working with large datasets and was controlled by tiling input files and saving intermediate assets, ensuring that only one large object was held in memory at a time.
- **Over-complex or slow steps** — some methods, such as the height-based cluster-growing algorithm and the texture-driven vegetation classifier, proved too demanding. To protect the project schedule, I moved these tasks to the future work list.

Risk	**Likelihood**	**Impact**	**Mitigation Strategy**
Technical Challenges in Implementation	Medium	High	Continuous self-study and leveraging online resources to keep up with technical advancements.
Over-complex or slow steps	Medium	High	Assess likelihood of completion and if needed move it to future work to protect the schedule.
Cost Constraints (RWT Asset)	Medium	Low	Opting for cost-effective alternatives and focusing on open-source solutions.
Delays in Development	Medium	High	Iterative rapid development processes.
Language Compatibility	Medium	Medium	Ensuring code compatibility across different programming languages like C# and Python. Regular code reviews and testing in the Windows environment.
Code Documentation and Recall	Medium	High	Implementing thorough documentation practices and regular code review sessions to ensure clear understanding and recall of the code's purpose and functionality.
Health Issues (Winter Period)	High	Medium	Flexible work schedules, and contingency planning for potential absenteeism.
Loss of Data/-Code/Project	Low	High	Regular backups and use of version control systems: Git and Unity Version Control.
Procrastination	Medium	High	Adherence to a structured Gantt Chart plan for task management and progress tracking.

Table 3.1: Risk Assessment

- **Winter illness** — to prepare for possible interruptions, I brought forward reading and design activities so that development could pause if needed without stalling overall progress.

Version control systems and careful use of open-source licences kept other risks, such as data loss or legal conflicts, within acceptable limits.

3.5 Testing

Testing had to cover two layers: the C# back-end (built in Visual Studio) and the Unity editor, whose frequent recompiles and asset refreshes strain memory.

White-box unit tests - Every core class has Edit-Mode tests in Unity Test Runner to catch logic errors and regressions.

Black-box checks - Added as soon as a pipeline step ran end-to-end; each check appears beside the step it protects in Chapter 4.

Visual checks - Regular scene inspections confirmed that meshes, terrain, aerial imagery, and map layers lined up. Because coordinate drift is common in Unity, the human eye—my "wetware" test bench—remained vital.

Automating that last stage is left for future work.

3.6 Reflections

Choosing an Unconventional Project

I deliberately designed this project to explore an unconventional direction. Rather than following typical game development pathways, I chose to focus on building a flexible design tool within a commercial game engine. I was particularly interested in working with LiDAR data, which is usually applied in surveying and research rather than creative design. From the start, I knew this would be an experimental process: the outcome was uncertain, and each stage revealed new challenges and layers of complexity. This uncertainty was part of the appeal, offering the opportunity to create something original and to solve problems that few had tackled in this context.

Personal progress

Early enthusiasm (*How hard that can be ?*) made the challenge look manageable. As soon as the pipeline touched million-point files, reality set in: parser speed, memory limits, and coordinate mismatches demanded fresh fixes almost weekly.

Dropping over-ambitious tasks kept the project alive and taught me to design smaller experiments with clear exit points.

Testing in practice

White-box tests handled silent errors, but final approval remained visual. If the scene did not *look* right, the code had to change. Image-based comparison or scripted camera paths could reduce this manual step in future versions.

Evaluating the result

Checking how well the tool works was itself difficult. The project sits in a small niche: few people try to drive raw LiDAR through a game engine, and those who do often have far more equipment and staff. I had no published benchmark that matched my scale, no hardware powerful enough to run city-wide tests, and no completed application on which to gather user feedback through a focus group.

Instead I judged progress against my own goals. The tool does load the point cloud, keeps data in point form until the last moment, applies a basic classifier, and places editable meshes in Unity—that covers goals G1 to G4. Goal G5, which asks for faster converters and better texturing, remains open.

These limits are not a failure; they simply mark where the first version stops. Any further work will need clearer metrics, wider test areas, and perhaps a small user study once the interface is stable enough to share.

On writing the report

Because development and discovery ran together, the report could not wait for a "finished" prototype—there was no such clear point. The writing therefore took the form of a journal: decisions, tests, and fixes are recorded in the order they happened. That is why reflection appears here, earlier than in many dissertations: the reader deserves to know why some parts are polished, others experimental, and a few postponed altogether. Time finally ran out, and the prototype stopped at a working but limited state. The steps that follow in Chapter 4 show what was achieved.

Chapter 4

Implementation

This chapter presents a practical walk-through of the system pipeline that I developed, starting with early terrain experiments and advancing through data conversion, classification, clustering, mesh generation, and finally first-person navigation (with references to derived use cases discussed further in Chapter 5).Each step follows a **Problem → Solution → Test → Comment** format and includes illustrative figures.

4.1 Initial Setup and GameManager Configuration

Problem

I needed a minimal Unity scene with configurable data paths to drive the entire processing pipeline via an editor GUI.

Solution

- I created a new Unity scene containing only a `Camera`, a `DirectionalLight`, and an empty `GameObject` named `GameManager`.
- I attached my custom `GameManager` script to expose all input/output paths and pipeline parameters in the Inspector.
- I prepared raw assets under `Assets/Data`:
  - `su4828_I2_250_03flip.jpg` (aerial image)

- `su4829.png` (InfoMap topographic image)
- `su4829.laz` (raw LiDAR point cloud)
- `su4829_DTM_50CM.asc` (ASCII DTM)

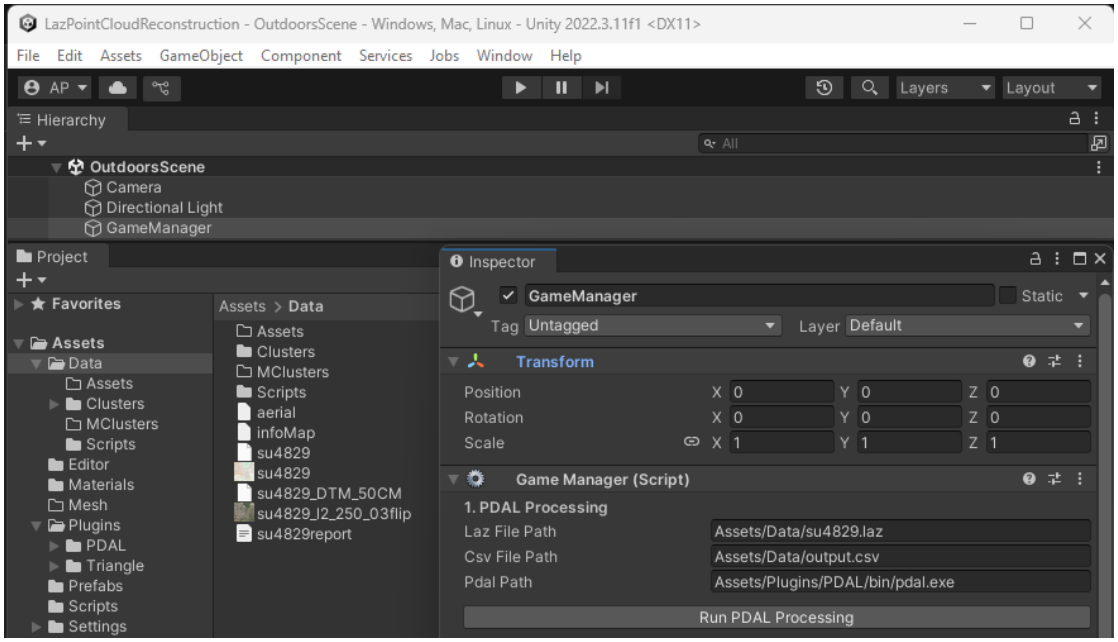

FIGURE 4.1: Initial `GameManager` configuration showing PDAL paths and data inputs.

Test

I verified that all file paths loaded correctly. As shown in Figure (4.1, the Inspector for GameManager displayed the PDAL input/output paths: the LAZ file, the target CSV output file, and the location of the PDAL executable. These values were automatically filled using hardcoded defaults defined in the GameManager script.

Comment

This setup centralized control and allowed me to execute each pipeline stage with a single click from the Unity Editor.

4.2 Step 1 – Terrain Height-map Experiment (First Iteration)

Problem

I aimed to quickly display real-world elevation in Unity with minimal custom code, to validate heightmap import and first-person exploration.

Solution

In this first iteration, I:

- Downloaded DTM and DSM ASCII files (50 cm resolution) from EDINA Digimap for grid SU4829 (Winchester) and related topographic and aerial images [7, 6, 8, 12].
- Converted ASCII to 16-bit RAW heightmaps using GDAL after previewing in QGIS (Figure 4.2, Table 4.1).
- Created two Unity Terrain objects (1,000 m × 1,000 m, 2,000×2,000 resolution), using the DTM for base topography and the DSM for surface features.
- Overlaid a rescaled topographic map (2000×2000, padded to 2048×2048) on the DTM (Figure 4.3) and a rescaled aerial image (4096×4096) on the DSM (Figure 4.4).
- Added Unity's standard CharacterController for first-person movement (WASD, mouse look, Q/E rotation, Z/X pitch, Ctrl+Q exit) (Figure 4.5).

TABLE 4.1: Statistics of DTM(su4829_DTM_50CM.asc) and DSM (su4829_DSM_50CM.asc) Files

Statistic	**DTM**	**DSM**
Minimum Value	32.516	12.935
Maximum Value	90.356	110.206
Mean Value	43.194	45.659
Standard Deviation	12.561	12.903
Valid Percent	100%	99.67%
Range	57.84	97.271

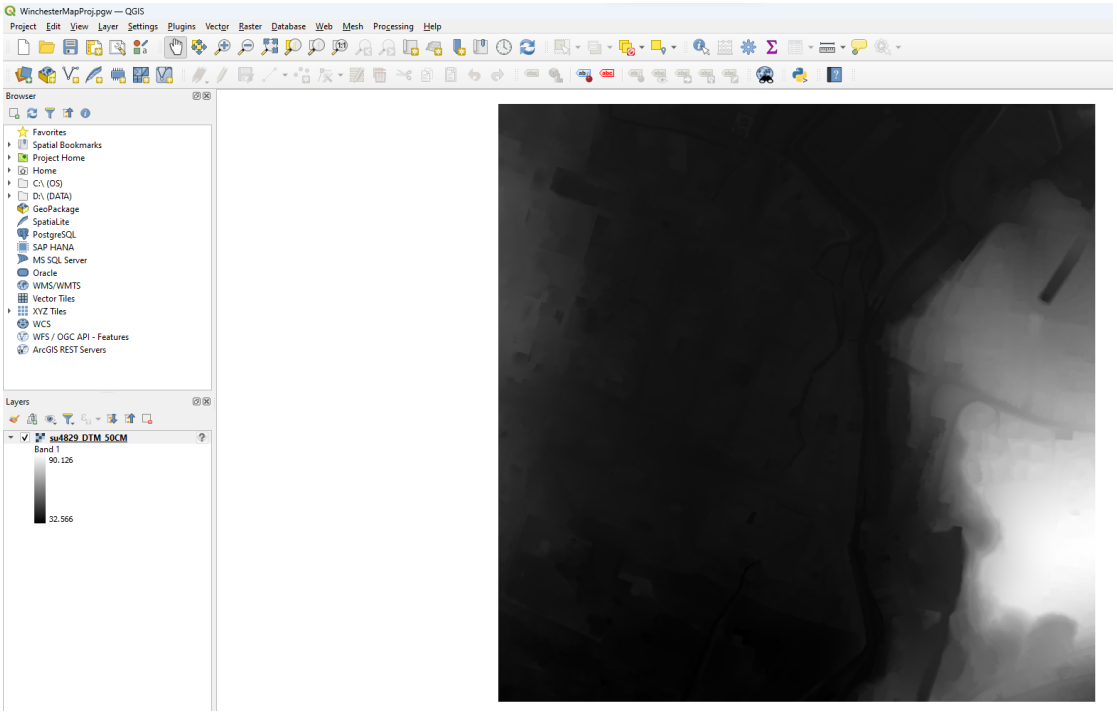

FIGURE 4.2: Preview of DTM and DSM data in QGIS before processing

Test

I overlaid the topographic map onto the DTM terrain and confirmed accurate feature alignment (Figure 4.3). The first-person controller allowed smooth, eye-level navigation, validating the interaction mechanics. When I applied the aerial image to the DSM terrain (Figure 4.4), distant landmarks appeared correct, but close inspection revealed blurred building edges and no vertical walls.

Comment

The successful DTM topography overlay and first-person navigation were retained and reused in later stages (e.g., terrain validation in Section 4.14). The shortcomings of the aerial-DSM overlay—blurred details and lack of vertical geometry—confirmed (Figure 4.5)the need to move to a point-cloud–based approach.

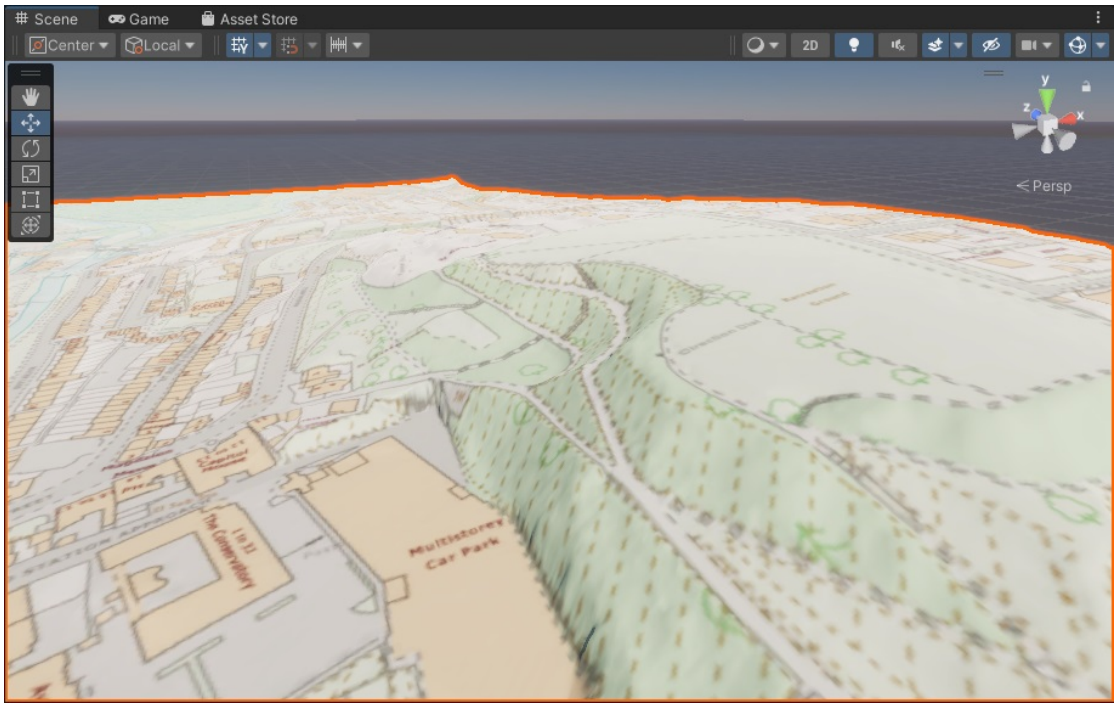

FIGURE 4.3: Topographic map overlaid on DTM terrain

4.3 Step 2 – Height-Based Region Growing

Problem

I needed to isolate individual buildings using only the 50 cm DSM data.

Solution

- Computed DSM–DTM difference to highlight above-ground objects.
- Seeded a region-growing algorithm at the highest DSM point (Winchester Cathedral tower) with a 4 m vertical threshold.
- Visualized results as cuboids and voxels in Unity.

Test

The cuboids matched the cathedral footprint (Figure 4.6); voxels exposed horizontal misalignment due to coordinate mismatches (Figure 4.7).

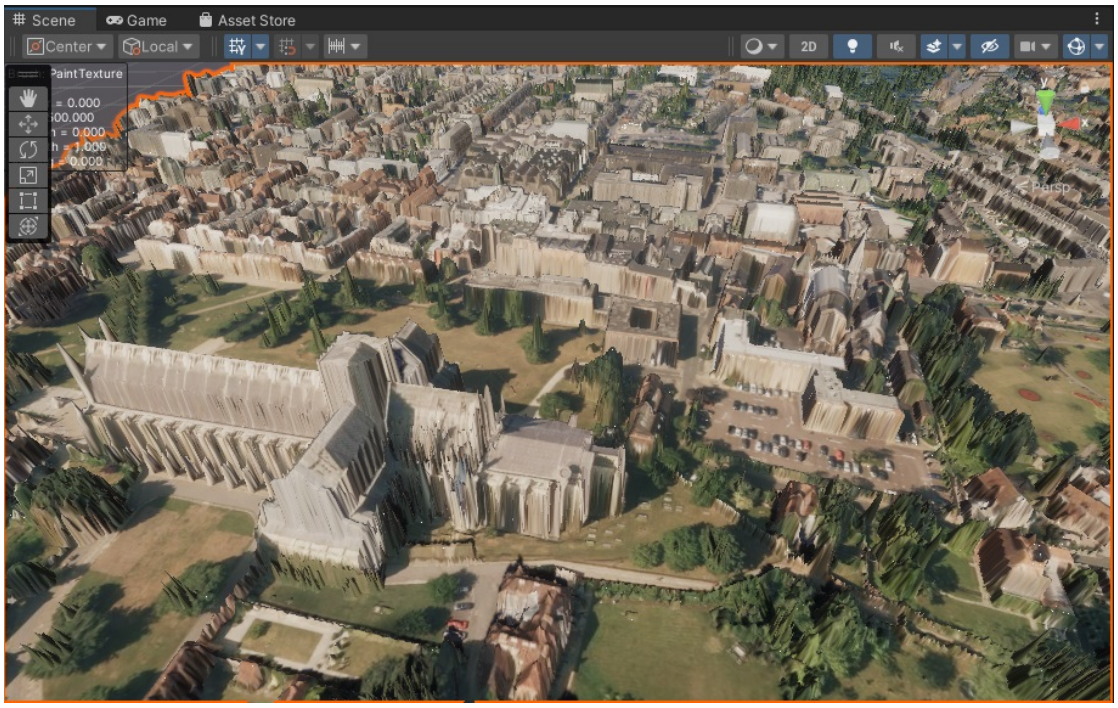

FIGURE 4.4: Aerial imagery overlaid on DSM terrain

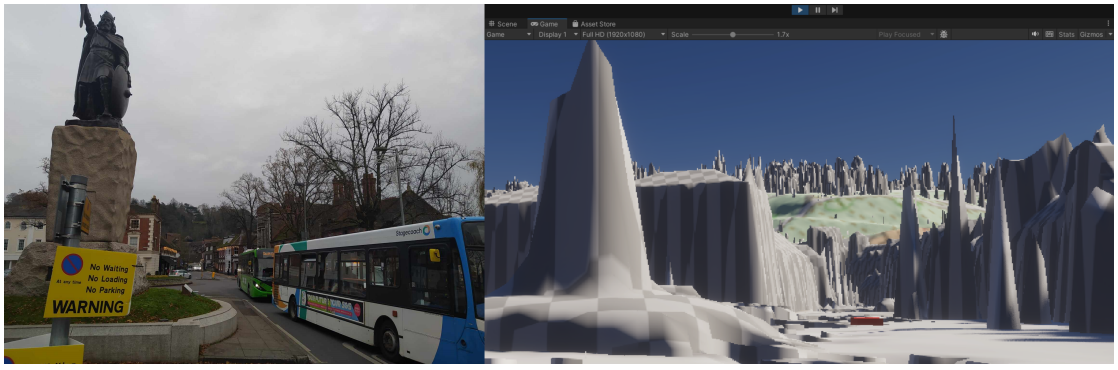

FIGURE 4.5: Comparison of real-world features and Unity scene (Winchester City Mill)

Comment

Region-growing worked conceptually, but coordinate reprojection issues forced a full point-cloud pipeline.

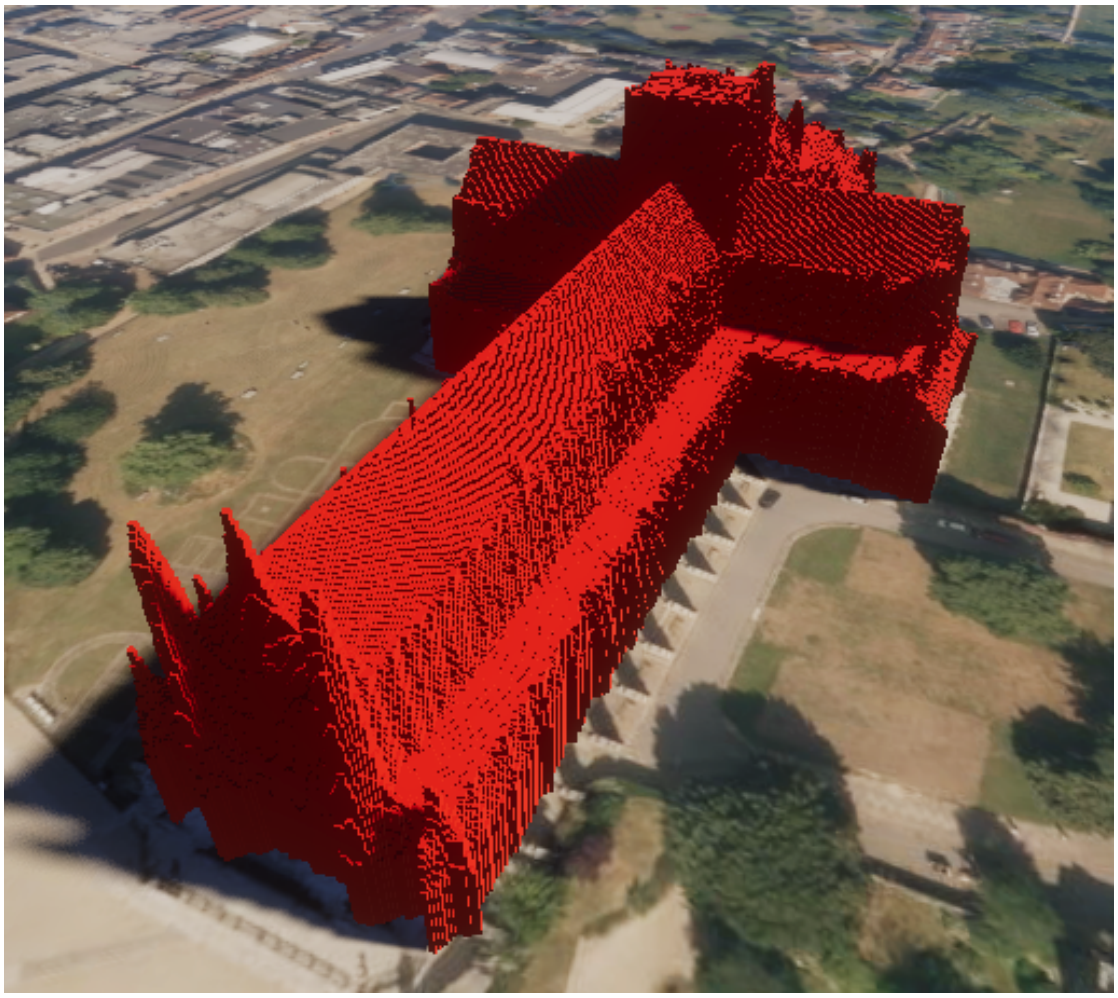

FIGURE 4.6: Cuboid outline of Winchester Cathedral

4.4 Step 3 – Convert LAZ to CSV

Problem (Use Cases 1.1–1.3)

I needed to convert the raw LAZ file to CSV and handle errors.

Solution

- Clicked `Run PDAL Processing` to execute `pdal.exe translate input.laz output.csv`.
- Trimmed the CSV to required fields.
- Provided fallback for manual CSV assignment.

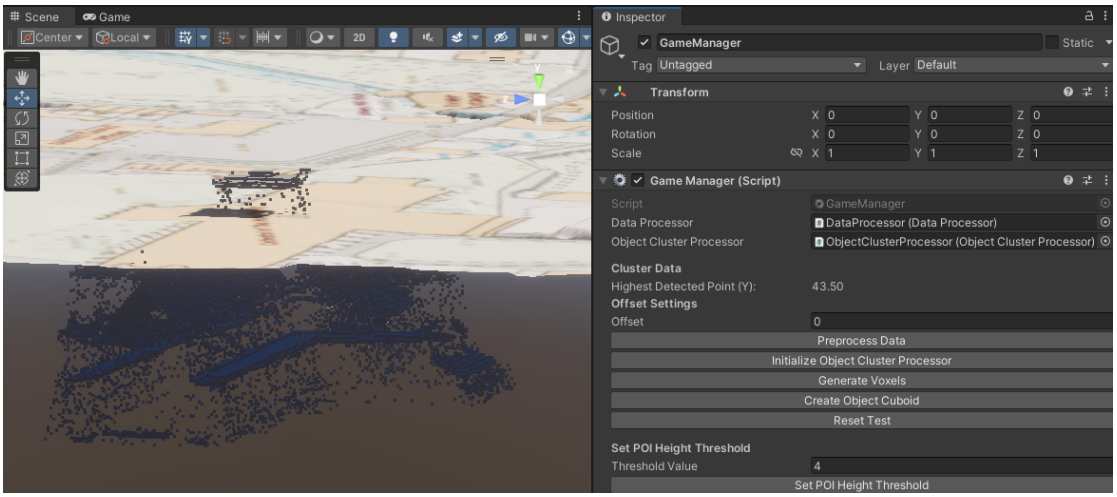

FIGURE 4.7: Voxel visualization highlighting horizontal misalignment

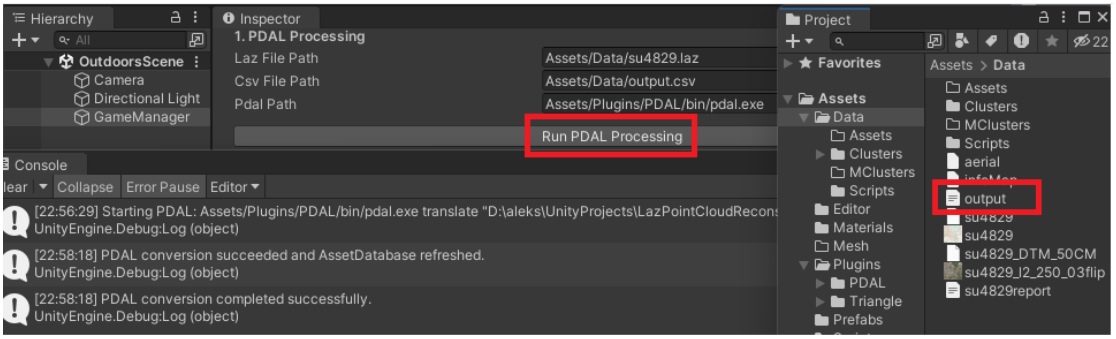

FIGURE 4.8: Converted LAZ to CSV using PDAL via the editor GUI

Test

Confirmed `output.csv` under `Assets/Data` and PDAL logs in Console. (After pressing the Run PDAL Processing button in the Inspector window (as shown in Figure 4.8), the embedded pdal.exe tool executed a translation from the LAZ file to CSV. The resulting output.csv file was generated and appeared under the Assets/Data folder, confirming a successful conversion.) .

Comment

Reliable conversion with manual override ensured workflow continuity.

4.5 Step 4 – Assign Vegetation Texture

Problem (Use Cases 3.1–3.2)

I needed to attach and preview a vegetation-classification texture for later point filtering.

Solution

I dragged `su4828_I2_250_03flip.jpg` into the `Vegetation Texture` slot in the Inspector. I then clicked `Test Texture Statistics` to log pixel statistics, including minimum/maximum RGB values and quadrant-specific averages (as shown in Figure 4.9).

Test

Logged statistics confirmed the correct orientation and colour ranges, validating the texture's suitability for classification (Figure 4.9).

Comment

Pre-flipping the image offline (e.g., using Paint) was more efficient than transforming pixels at runtime.

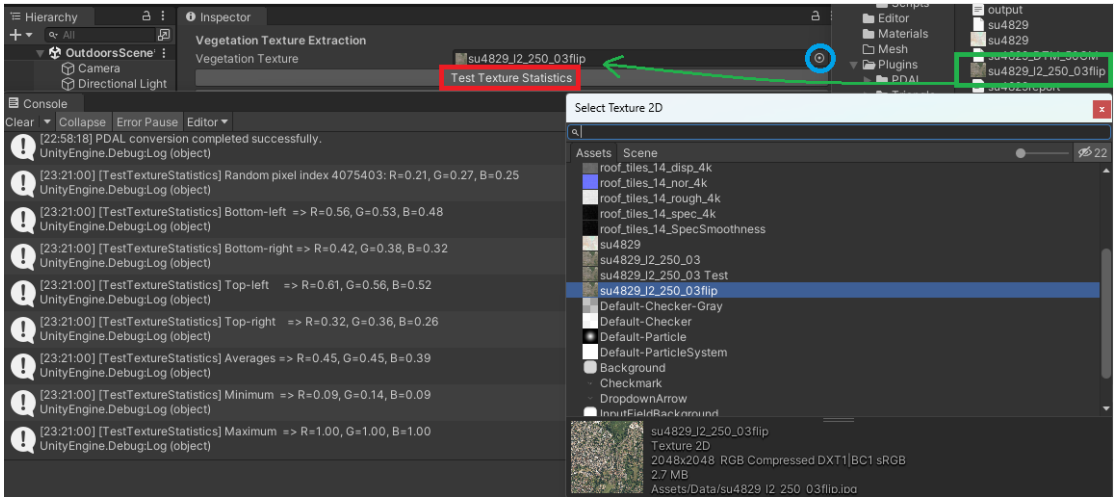

FIGURE 4.9: Assigned vegetation texture and tested pixel statistics.

4.6 Step 5 – Configure CSV and DTM Inputs

Problem (Use Cases 2.1–2.4)

I needed to configure CSV, DTM, and grid subdivision parameters before creating ScriptableObjects.

Solution

I assigned `output.csv` and `su4829_DTM_50CM.asc` in the Inspector. The output asset path was set to `PointDataCollection.asset`, and the expected point count field was populated. I selected a grid divider value of 8, subdividing the area into 64 sectors (as shown in Figure 4.10).

Test

Default values were verified, and the sector count preview confirmed successful subdivision (Figure 4.10).

Comment

Sectoring prevented memory spikes during full dataset processing, improving performance stability.

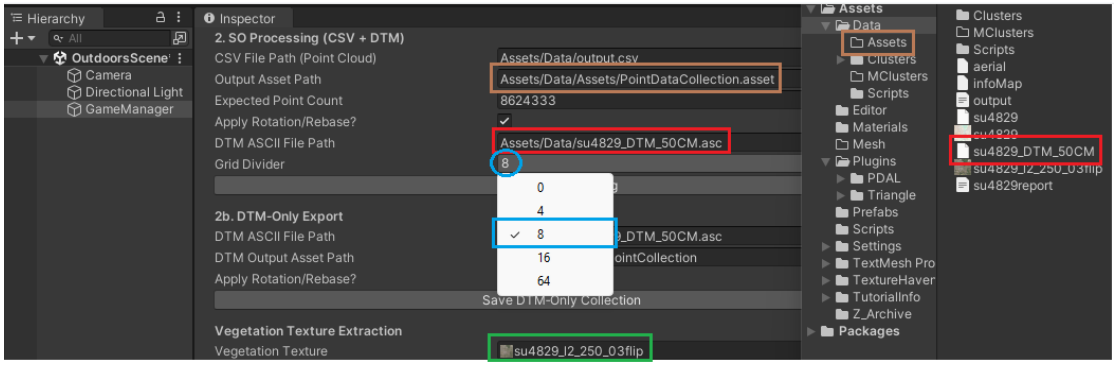

FIGURE 4.10: Configured CSV, DTM, and grid for SO processing.

4.7 Step 6 – Run SO Processing and Save Collections

Problem (Use Cases 2.1–2.4)

I needed to serialize the point cloud with DTM height information into ScriptableObjects, and optionally export DTM-only assets.

Solution

I clicked `Run SO Processing` to generate the `PointDataCollection.asset`. Additionally, I clicked `Save DTM-Only Collection` to export the DTM sample alone (as shown in Figure 4.11).

Test

The Console confirmed that 64 sector assets and the DTM-only asset were successfully created (Figure 4.11).

Comment

Automating the DTM sampling and sector division process reduced manual intervention and ensured reproducibility.

4.8 Step 7 – Extract Classifier Collection

Problem (Use Cases 4.1–4.3)

I needed to filter points by LAS classification and vegetation attributes.

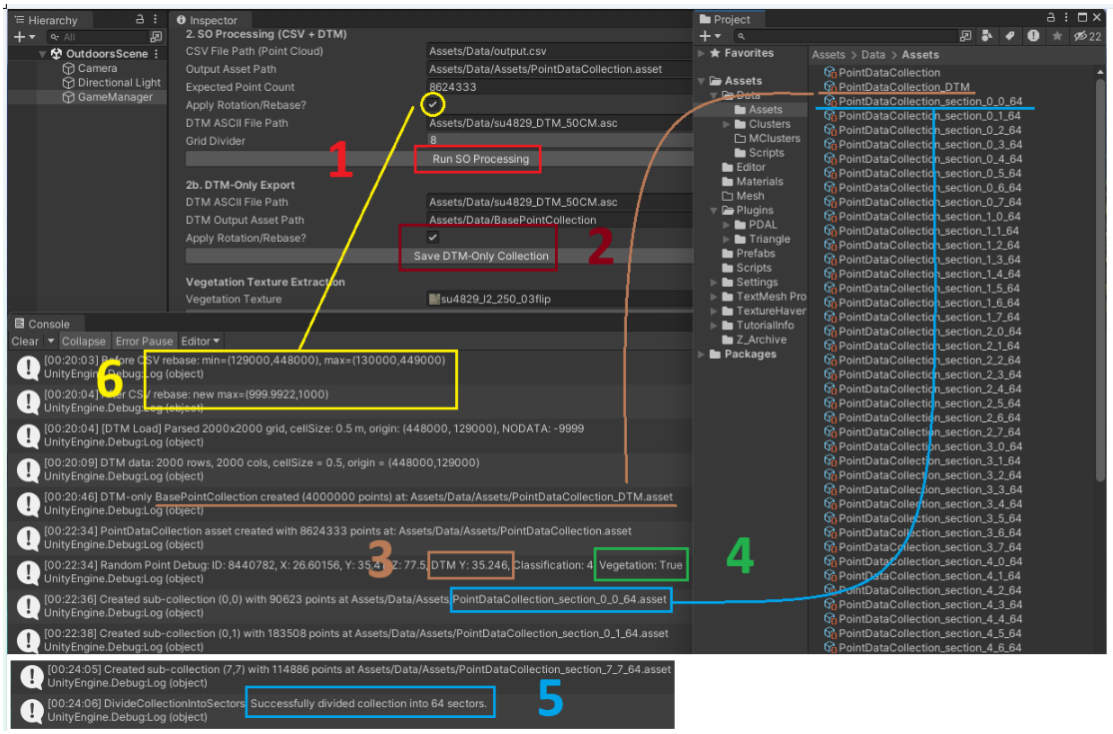

FIGURE 4.11: Ran SO processing: DTM sampling and sector division.

Solution

I selected sector (4,2), set the classifier filter to "high vegetation" (value 5), enabled the "Only Non-Vegetation" option, and clicked `Extract Classifier Collection` (as shown in Figure 4.12).

Test

The resulting `Object_high.asset` contained 38,478 points, matching expectations (Figure 4.12).

Comment

Filtering significantly reduced data volume, which improved processing time for later stages.

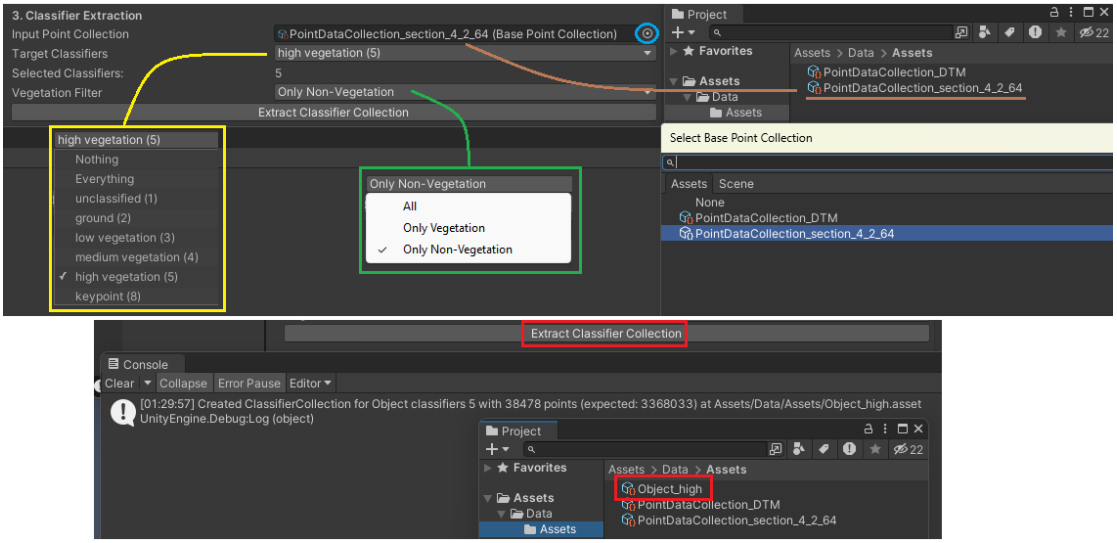

FIGURE 4.12: Filtered by class and vegetation to extract relevant points.

4.9 Step 8 – Create KDTreeCollection Asset

Problem (Use Case 5.1)

I needed a KD-tree structure to optimise spatial neighbour queries for clustering.

Solution

I selected `Object_high.asset`, kept the default number of neighbours at 10, and clicked `Create KDTreeCollection Asset` (as shown in Figure 4.13).

Test

The Console output confirmed that 10 neighbours were calculated for each point (Figure 4.13).

Comment

Precomputing neighbours drastically improved later clustering performance.

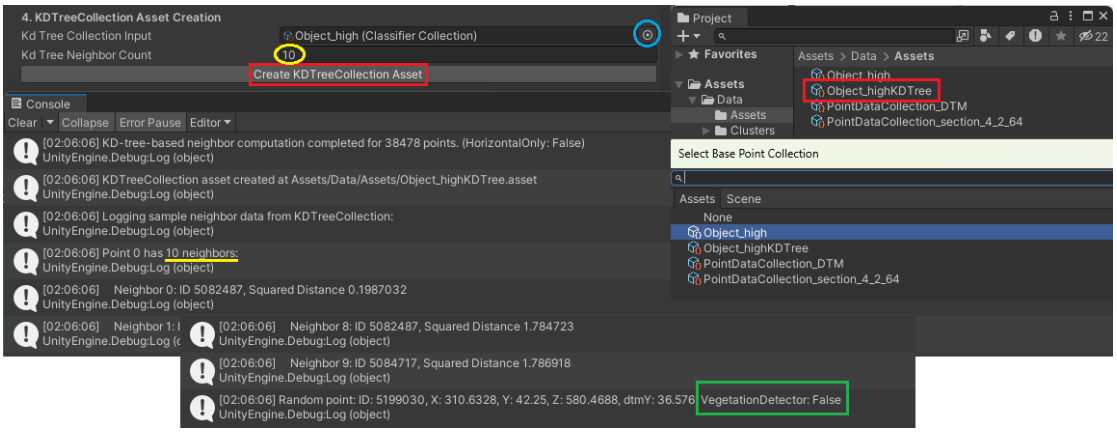

FIGURE 4.13: Generated KD-tree and verified neighbour logs.

4.10 Step 9 – Clustering

Problem (Use Cases 6.1–6.3)

I needed to group points into clusters based on spatial proximity thresholds.

Solution

I set the clustering threshold to 0.7m, minimum points per cluster to 1,000, and maximum points to 100,000, then enabled `Save Clusters` and `Save Cluster List` (as shown in Figure 4.14).

Test

556 clusters were initially found, with 12 clusters saved after applying size thresholds (Figure 4.14).

Comment

Saving a master cluster list simplified batch mesh generation and improved project organisation.

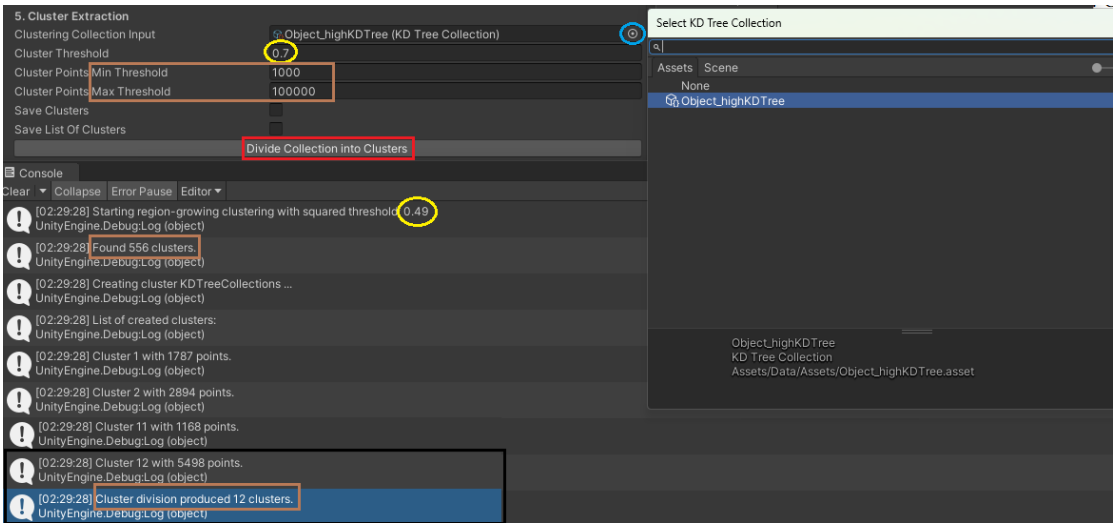

FIGURE 4.14: Region-growing clustering and size filtering.

4.11 Step 10 – Mesh Generation

Problem (Use Cases 7.1–7.3)

I needed to generate simplified meshes from clustered point groups.

Solution

For each cluster in the list:

- Projected points to the XZ-plane and applied Delaunay triangulation (using Triangle.NET),
- Computed 2D boundaries via an R-table method,
- Extruded boundary points downward based on DTM heights (as shown in Figure 4.15).

Test

Meshes were successfully generated and saved under `Assets/Mesh/ClusterGroup_0` (Figure 4.15).

Comment

Meshes were lightweight, edit-friendly, and correctly aligned with the original terrain.

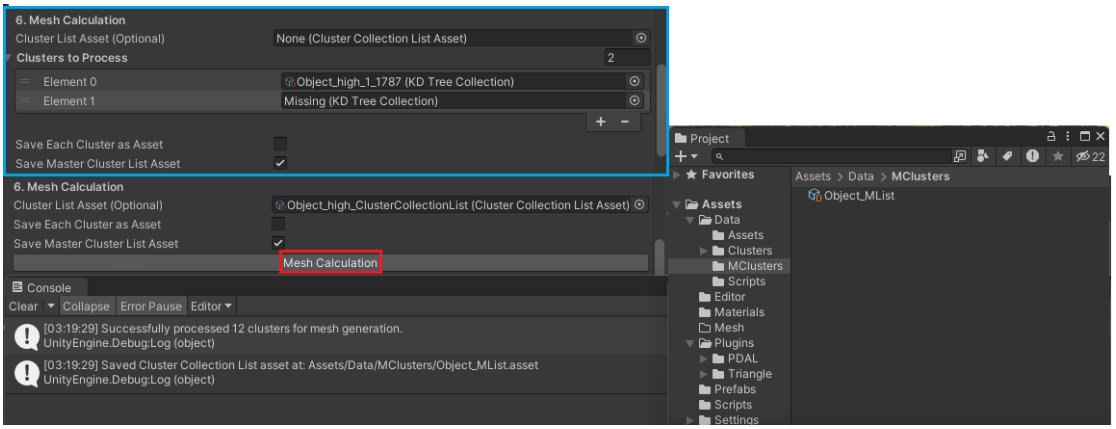

FIGURE 4.15: Mesh generation from clustered points using triangulation and extrusion.

4.12 Step 11 – Model Mesh Building

Problem (Use Cases 8.1–8.3)

I needed to instantiate the generated cluster meshes as GameObjects in the Unity scene.

Solution

I selected the cluster mesh list and clicked `Model Mesh Building`, which created GameObjects with Roof and Body child meshes (as shown in Figure 4.16).

Test

Mesh groups were successfully instantiated in the scene hierarchy, grouped according to their source list (Figure 4.16).

Comment

Structured GameObject hierarchy made the scene cleaner and easier to navigate.

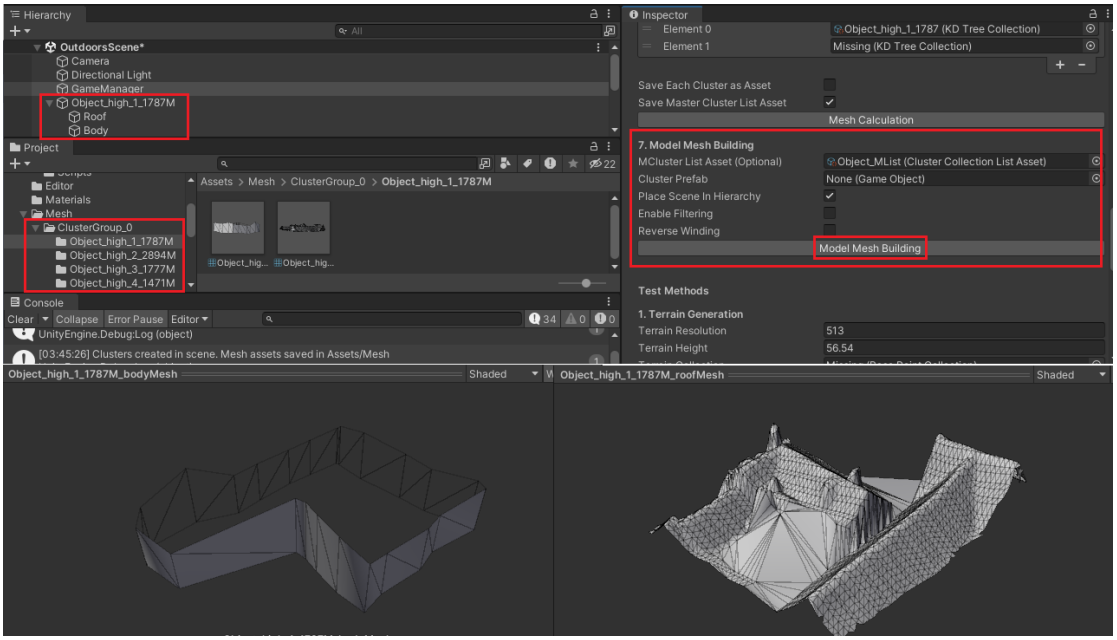

FIGURE 4.16: Instantiated cluster meshes and grouped assets in the Unity scene.

4.13 Step 12 – Locating Clusters in Scene View

Problem

I needed to quickly locate specific clusters in the Unity scene for debugging.

Solution

By double-clicking either the Roof or Body child GameObject, I could quickly frame the corresponding cluster in Scene view (as shown in Figure 4.17).

Comment

This technique significantly improved efficiency during scene inspections.

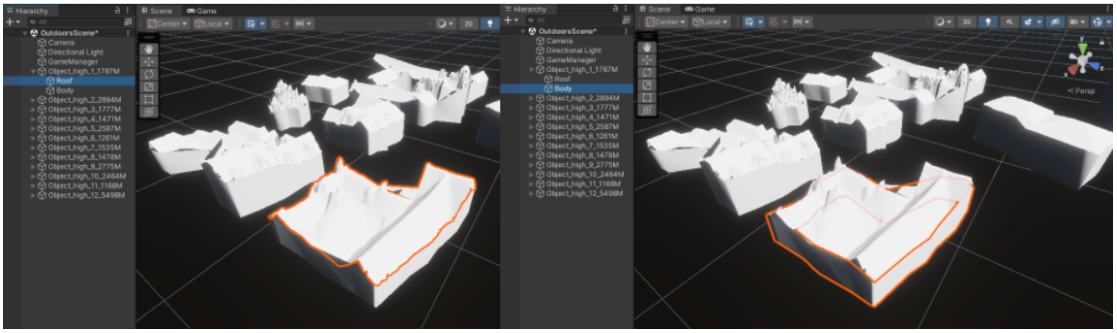

FIGURE 4.17: Locating individual cluster meshes by double-clicking in the Scene view.

4.14 Step 13 – Terrain Generation

Problem (Use Cases 9.1–9.3)

I needed to generate a Unity Terrain object from the sampled DTM data.

Solution

I assigned the DTM-only asset, clicked `Compute Terrain Stats`, then clicked `Create Terrain` (as shown in Figure 4.18).

Test

The resulting Terrain object aligned to `y=0` and matched the expected real-world profile (Figure 4.18).

Comment

The terrain provided a stable reference layer for validating the reconstructed meshes.

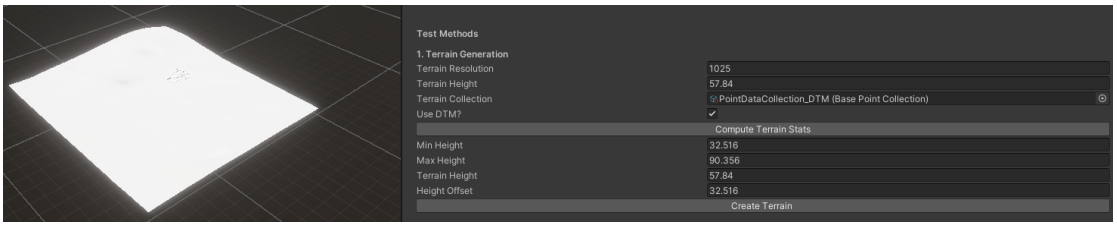

FIGURE 4.18: Generated Unity Terrain object based on DTM sampling.

4.15 Step 14 – Terrain Texturing and Validation

Problem

I needed to overlay a topographic InfoMap texture to verify mesh placements.

Solution

I applied the `su4829.png` texture as a Terrain Layer and adjusted tiling settings (as shown in Figures 4.19 and 4.20).

Test

Cluster positions visually matched real-world features like Winchester City Mill (Figures 4.19 and 4.20).

Comment

Overlaying a real map helped visually validate mesh and terrain alignment.

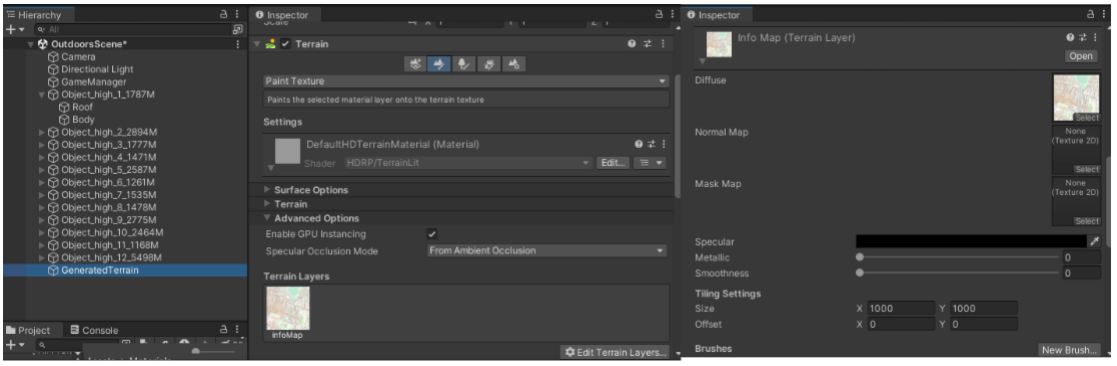

FIGURE 4.19: Applied topographic InfoMap layer to generated terrain.

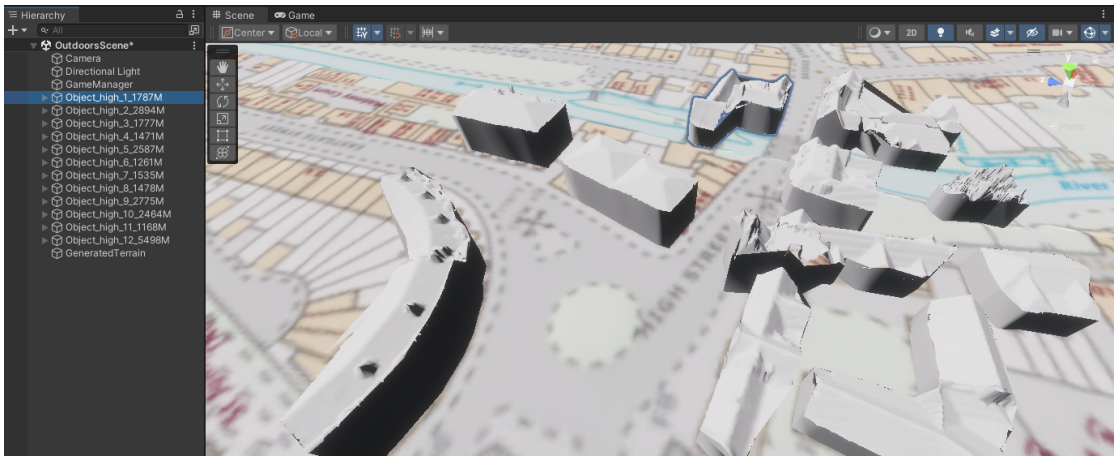

FIGURE 4.20: Visual validation of mesh placement using InfoMap overlay.

4.16 Step 15 – First-Person View Test

Problem

I needed to explore the scene from a first-person perspective.

Solution

I attached the `Init.cs` player controller script to the terrain and configured movement controls (as shown in Figure 4.21). I then entered Play mode and explored the reconstructed scene at ground level (as shown in Figure 4.22).

Comment

First-person navigation confirmed that the environment was both correctly aligned and usable.

4.17 Summary

I encapsulated the complete pipeline—from raw LiDAR through CSV, classification, clustering, mesh generation, scene assembly, terrain creation, to first-person

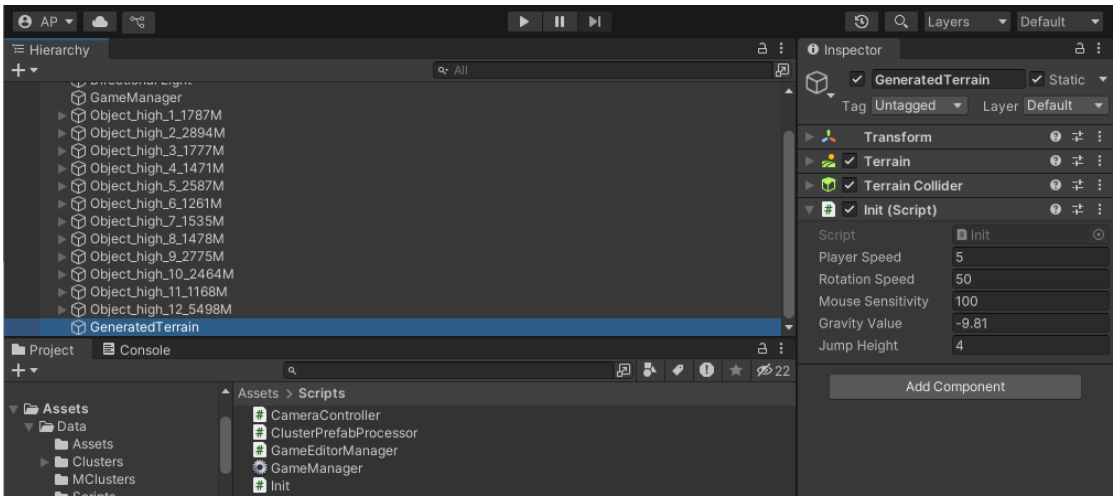

FIGURE 4.21: Attached first-person controller to terrain.

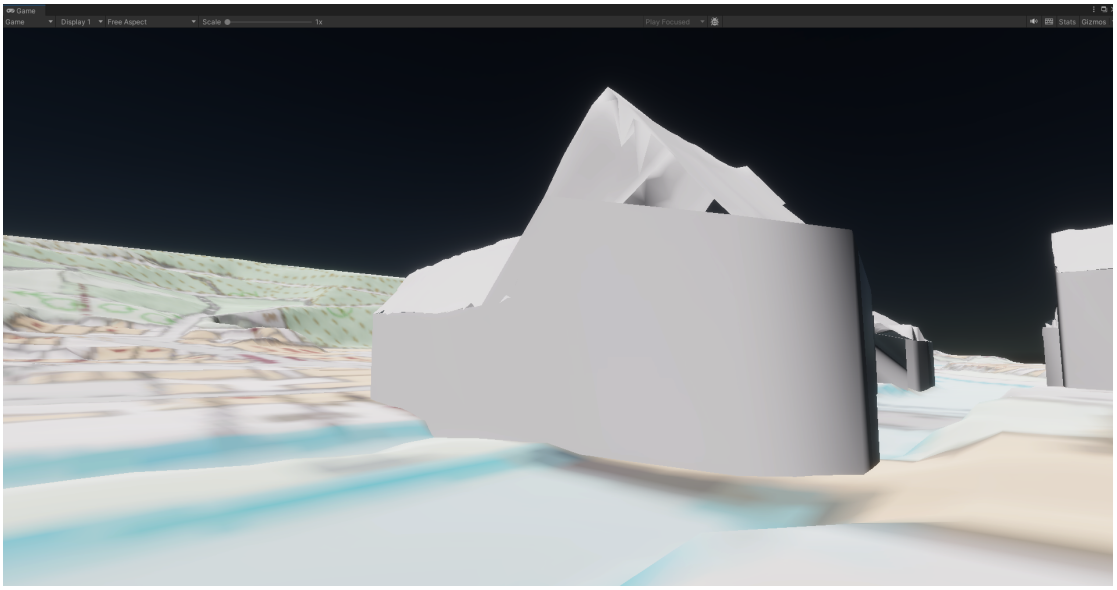

FIGURE 4.22: Navigated the reconstructed scene in first-person mode.

testing—using Unity Editor tools. Each step can be rerun independently or recombined for alternate workflows. Future enhancements include performance optimization, richer texturing, and full automation (see Chapter **??**).

Chapter 5

Design

5.1 Overview

Through trial, research, and refinement I arrived at a robust pipeline—one that handled real-world LiDAR and DTM data reliably, extracted meaningful clusters, and generated editable mesh models entirely within Unity. This design chapter describes the outcome of resolving the challenges identified in Chapter 3 (scope, goals, risks) and the solutions implemented in Chapter 4 (terrain import, classification, clustering, meshing). Figure 5.1 summarizes the integrated workflow I developed.

5.2 Use Cases

The following use cases emerged directly from the experiments in Chapter 4. They document each discrete processing step I exercised and form the foundation for the formal requirements that follow. While these were born out of development, they also serve as a checklist for future testing (see Chapter 6).

Table 5.1: Pipeline use case table

Use Case	Description
1. Convert LAZ to CSV	

Use Case	**Description**
1.1 Convert LAZ to CSV using PDAL	Run PDAL translation using embedded executable and log outputs.
1.2 Trim CSV to only required fields	Optimise file size by excluding redundant attributes.
1.3 Handle PDAL errors and fallbacks	Ensure fallback if PDAL fails; allow user to provide CSV manually.
2. Serialise CSV to ScriptableObject	
2.1 Serialise base point collection from CSV	Read and store structured data from CSV into a Unity asset.
2.2 Serialise DTM data to Base Point Collection (from ASCII)	Create point collection based solely on ASCII DTM data.
2.3 Integrate terrain heights into point cloud	For each point in the cloud, compute its DTM grid indices (via floor(point.x/cellSize), etc.), look up the corresponding elevation, and assign it to the point's `dtmY` field.
2.4 Divide into sectors (optional)	Split large area into square regions to reduce processing time.
3. Add Vegetation Flag	
3.1 Add vegetation flag from serialised data using 2D texture	Run pixel-based classifier on 2D texture to mark vegetation.
3.2 View vegetation overlay in Unity scene	Visualise vegetation points using coloured debug cubes. When used together with terrain generated from aerial imagery, this helps test placement accuracy.
4. Filter by Classifier	
4.1 Extract subset by classification value	Select specific classes like ground or high vegetation from the dataset.
4.2 Filter by custom enum flag	Enable selection of vegetation, non-vegetation, or all points.

Use Case	**Description**
4.3 Chained filtering with previous outputs	Re-apply filters on already filtered datasets for fine-tuned control.
5. Create KD-Tree	
5.1 Generate KDTreeCollection from Base Point Collection	Build spatial structure for neighbourhood queries and clustering.
5.2 Create multiple KD-trees	Test using different filtered datasets.
5.3 Use KDTree with terrain generation	Enable point-based terrain visualisation.
6. Divide into Clusters	
6.1 Run cluster test with threshold logging	Log number of clusters and how many are accepted.
6.2 Save accepted clusters as individual assets	Export each valid cluster as a separate Unity asset.
6.3 Save list of accepted clusters	Store accepted clusters in a single asset for batch processing.
7. Mesh Calculation	
7.1 Calculate mesh for individual cluster	Generate mesh using point cloud triangulation for a single cluster.
7.2 Calculate mesh for list of clusters	Batch generate meshes for multiple clusters.
7.3 Create mesh assets with cluster metadata	Store triangulation results with centroid and boundary data.
8. Build Model Mesh in Unity	
8.1 Add individual processed cluster to Unity scene	Instantiate mesh in Unity using precomputed geometry.

Use Case	Description
8.2 Add list of processed clusters	Populate the scene with a full dataset of models.
8.3 Reuse terrain-aware mesh placement logic	Ensure models are placed at correct elevation based on DTM.
9. Terrain Generation	
9.1 Serialise DTM data from ASCII	Generate a terrain-specific point collection used for accuracy testing.
9.2 Visual test of any collection using terrain	Allow visual inspection of data using Unity terrain features.
9.3 Generate terrain from DTM-focused collection	Output Unity terrain asset for placement verification.
10. Output	
10.1 Navigate using FPV camera	Walk around the environment using first-person view controller.
10.2 Edit or delete mesh in Unity scene	Remove or reposition mesh assets manually.
10.3 Verify placement and structure visually	Use visual debugging tools to confirm accuracy of processing.

5.3 Functional Requirements

Based on the initial requirements (R1–R4) and goals (G1–G5) set out in Chapter 3, and guided by the use cases exercised in Chapter 4, I derived a set of functional requirements—the core capabilities that the final design supports. Each requirement below maps directly to one or more use cases.

(R1) **LAZ→CSV conversion via PDAL**
Implements Use Case 1.1: translates LAZ to CSV, trims fields, and provides a manual-CSV fallback.

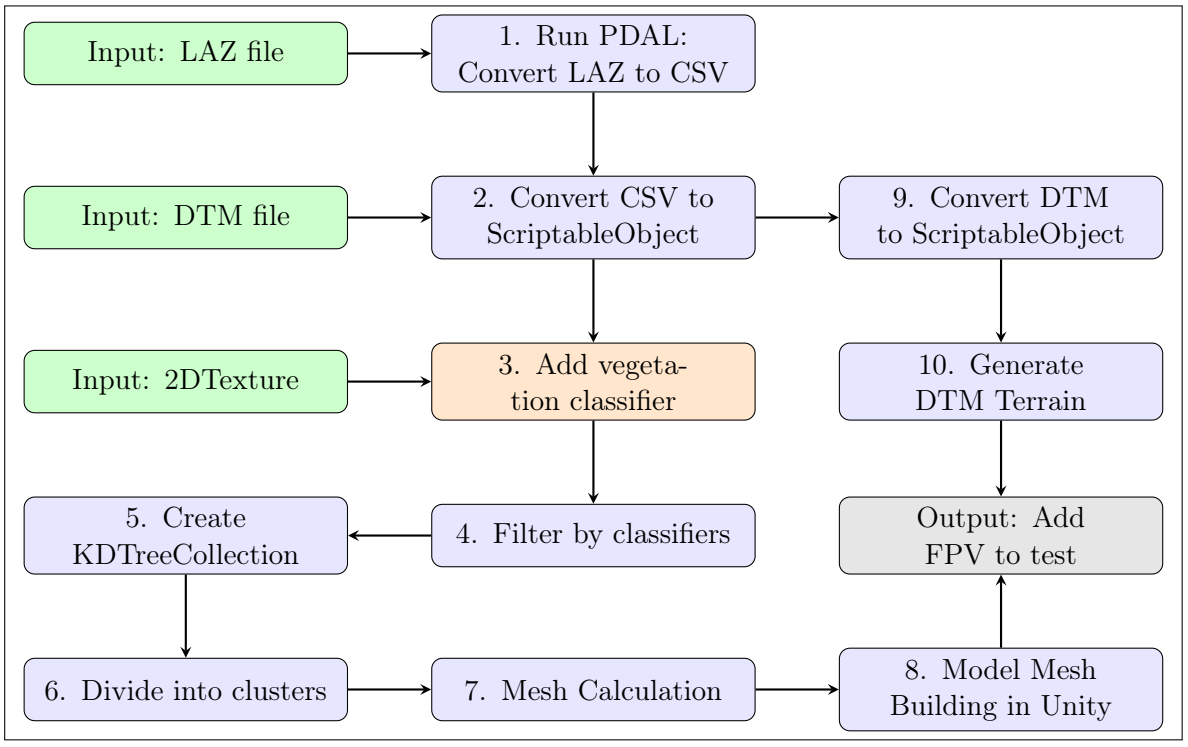

FIGURE 5.1: Functional pipeline design graph showing required input files, optional addition of vegetation classifier, and the final output mesh, which can be tested with a first-person view (FPV) script inside the Unity scene.

(R2) **CSV serialization to ScriptableObject**
Implements Use Cases 2.1–2.4: reads CSV, applies optional rotation/rebase, samples DTM heights, and subdivides into sectors to produce `BasePointCollection`.

(R3) **Vegetation flagging from texture**
Implements Use Case 3.1: applies a pixel-based classifier to flag vegetation, with QA logs.

(R4) **Classifier-based filtering**
Implements Use Cases 4.1–4.3: extracts subsets by LAS class and vegetation, reducing data size.

(R5) **KD-tree construction**
Implements Use Cases 5.1–5.3: builds `KDTreeCollection` assets with precomputed neighbours for fast queries.

(R6) **Distance-based clustering**
Implements Use Cases 6.1–6.3: groups points into clusters via region growing, filters by size, and saves cluster assets plus a master list.

(R7) **Mesh generation for clusters**
Implements Use Cases 7.1–7.3: triangulates roofs (Triangle.NET), computes boundaries (R-table), and extrudes walls to DTM height.

(R8) **Model instantiation in Unity**
Implements Use Cases 8.1–8.3: creates GameObjects from mesh data, groups them by batch, and positions them accurately.

(R9) **Terrain generation from DTM**
Implements Use Cases 9.1–9.3: generates a Unity `TerrainData` asset from ASCII DTM, computes stats, and aligns to y=0.

(R10) **First-person validation**
Implements Use Cases 10.1–10.3: adds the `Init.cs` controller for WASD/look navigation and visual QA.

All requirements were exposed through a single `GameManagerEditor.OnInspectorGUI()` interface, allowing modular execution without code recompilation.

5.4 Non-Functional Requirements

Compatibility and Portability

I developed the prototype on Windows 11 (Unity 2021.3 HDRP, Hub 3.11.1), using PDAL 2.8.4 and Triangle.NET. The tools run inside the Unity Editor; supporting other platforms requires platform-specific PDAL binaries.

Performance and Time

Measured on 1 km^2 tiles:

- CSV + DTM serialization: < 10 min.
- Classification, KD-tree, clustering: < 5 min.
- Mesh generation: 10–20 s per cluster (hours for a full tile).
- Terrain creation: seconds.

Reliability and Stability

I controlled memory via sectoring, classification filtering, and cluster size limits. Output assets were grouped into batch folders to avoid Unity's 5000-asset limit. PDAL failures triggered the CSV fallback. Large collections still risk memory spikes.

Tool Integration

- **PDAL 2.8.4:** LAZ→CSV conversion with timeout and logging.
- **Triangle.NET:** robust 2D triangulation for roof meshes.

Scalability

The prototype handled one 1×1 km tile at a time. Extending to multiple tiles requires enhanced memory management. Division by classifier and sector helped manage complexity.

Licensing

All third-party components (PDAL, Triangle.NET) are permissively licensed. The project is distributed as a Unity development project; no standalone installer is provided.

5.5 Data Flow and Structure

Because of memory constraints and the need for modular testing, I organised data into a series of serialised `ScriptableObject` assets. Each step in the Unity pipeline output its results as an asset that could be reused, inspected, or modified without needing to rerun the entire process. Figures 5.2 and 5.3 illustrate the overall structure of this system.

At the heart of both diagrams is the `KDTreeCollection` class, which played a central role in performance and data organisation. This class served as a bridge

between raw point data, neighbour computation, clustering, and mesh construction. It connected the spatial logic (e.g., neighbour queries, edge estimation) with geometric logic (e.g., mesh generation and wall extrusion), making it the key structure that bound together the modular stages. Its design enabled high performance during clustering (via precomputed neighbours) and offered a consistent format for passing data between tools.

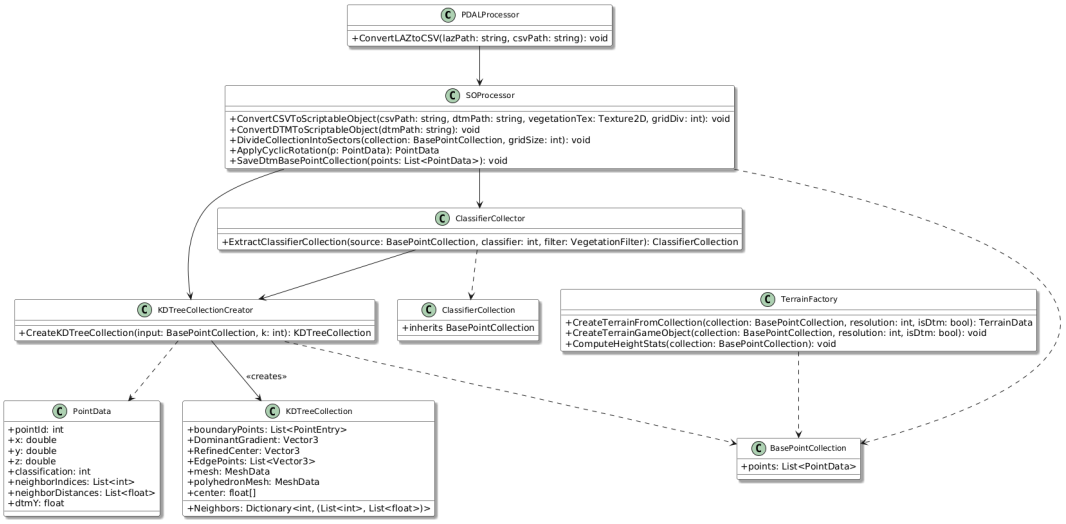

FIGURE 5.2: UML of data preparation: LAZ→CSV, DTM integration, classification, KD-tree.

FIGURE 5.3: UML of mesh processing: clustering, triangulation, extrusion, and model instantiation.

This design prevented Unity from overloading its memory and simplified debugging, since each ScriptableObject could be inspected and re-used independently.

5.6 Additional Comments

This design represents the culmination of exploratory experiments and incremental prototypes into a cohesive Unity toolset. It fulfilled goals G1–G4 (Chapter 3); remaining work (G5) will focus on performance tuning and advanced texturing.

Chapter 6

Testing

6.1 Introduction

This chapter presents the automated and manual testing approaches used to validate both the internal logic and the overall functionality of the pipeline. It comprises three complementary testing layers:

- **White-Box Unit Testing**: verifies core algorithms and data processing components at the code level.
- **Black-Box Scenario Testing**: executes end-to-end pipeline workflows against real use cases (BB9.x) to catch integration issues.
- **Visual ("Wetware") Testing**: relies on visual inspection within Unity's scene view to validate spatial alignment, mesh quality, and navigation—an essential practice given the complexity of geospatial reconstruction.

Automated tests are included here, while extensive visual examples that could not fit in this chapter are available in Appendix **??**.

6.2 White-Box Unit Testing

These tests were implemented in Unity's Test Runner (EditMode) using NUnit, directly derived from the use cases in Chapter 5. They guard internal logic against regressions.

6.2.1 Testing Strategy

- **Parser and Serializer (UC1.x)**: reading LAZ→CSV columns and CSV→ScriptableObje fields.
- **Vegetation Classifier (UC3.x)**: pixel-based detection under varied RGB inputs.
- **KD-Tree Construction (UC5.x)**: neighbour count and ordering checks.
- **Cluster Divider (UC6.x)**: region-growing behavior and size filters.
- **Mesh Processor (UC7.x)**: boundary extraction and extrusion correctness.

6.2.2 Key Failures and Fixes

- **UC112, UC113**: CSVReader initially processed 17 columns and parsed classification as float—fixed to four columns and integer parsing (Figure 6.1).
- **UC310–UC335**: synthetic vegetation texture tests revealed misclassifications under shadows, prompting parameter tuning (Figure 6.2).

6.2.3 More in Appendix

See the Appendix C

6.3 Black-Box Scenario Testing

These end-to-end tests (BB9.x) exercise the full pipeline steps as documented in Chapter 4, validating real workflows.

- **BB9.1**: Terrain height-map import and first-person walk-through (Pass).
- **BB9.2**: Height-based region growing output (Pass).
- **BB9.3**: LAZ→CSV conversion correctness (Pass).
- **BB9.4**: Vegetation texture assignment and stats (Pass).
- **BB9.5**: ScriptableObject creation with DTM and sectoring (Pass).

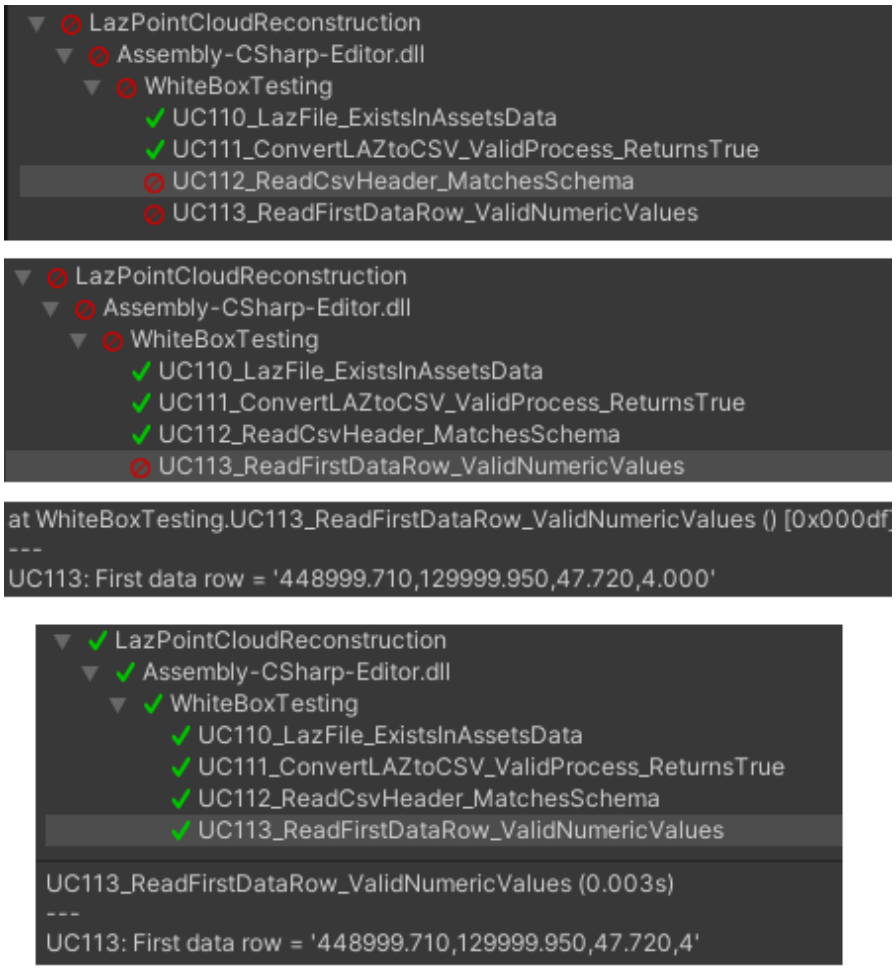

FIGURE 6.1: Unit test failures for CSV parsing and classification type.

- **BB9.6**: Classifier filtering and KD-tree build (Pass).
- **BB9.7**: Clustering with thresholds (Pass).
- **BB9.8**: Mesh generation for clusters (Pass).
- **BB9.9**: Scene assembly, terrain, and navigation (Pass).

6.4 Visual ("Wetware") Testing

Given the spatial and visual nature of the pipeline, manual inspection in Unity's Scene view was indispensable. Visual checks confirmed:

- Correct alignment of mesh clusters to DTM and textures.
- Absence of mesh artifacts or unintended connections (e.g., cathedral case study in Chapter 7).

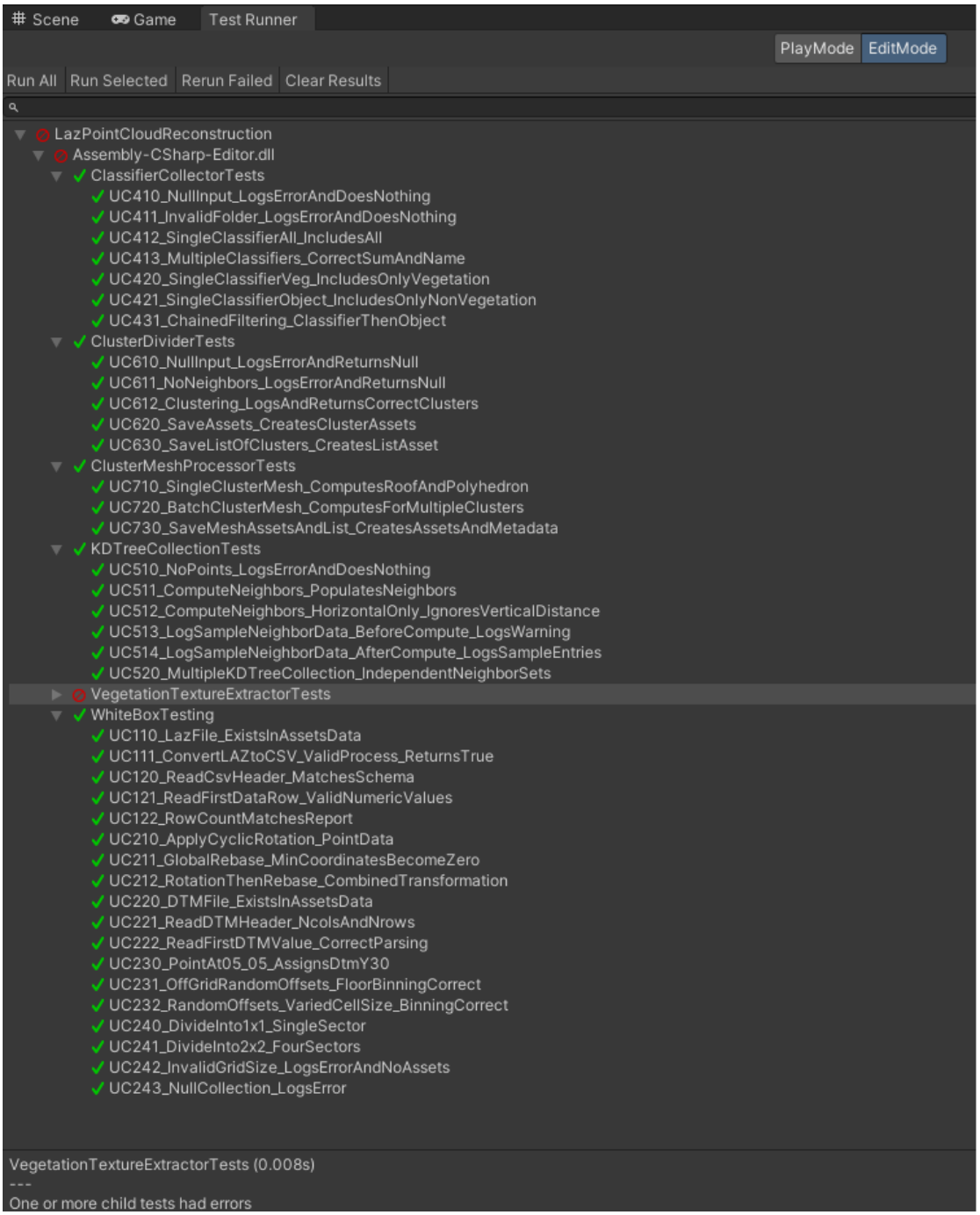

FIGURE 6.2: Vegetation classifier unit test results.

- Accuracy and quality of mesh artifacts.
- First-person navigation and impressions.

These wetware tests are informal but crucial; screenshots and annotations appear throughout Chapter 7, and a broader gallery is provided in Appendix **??** and Appendix D

6.5 Summary

The combination of white-box, black-box, and visual testing ensures both internal correctness and real-world usability of the pipeline. Unit tests rapidly detect logic regressions, scenario tests verify end-to-end workflows, and visual inspection safeguards the spatial fidelity that is central to this geospatial reconstruction tool.

Chapter 7

Evaluation

7.1 Introduction

This chapter evaluates the prototype against the functional and non-functional requirements defined in Chapters 3 and 5. It combines internal validation (against planned goals and use cases), a focused case study on a real-world structure, performance measurements, qualitative comparisons with related work, and lessons learned. Where relevant, I refer back to the discrete use cases (Chapter 5) and requirements (R1–R10) that guided development.

7.2 Functional Validation

7.2.1 Editable Unity Prefabs (R8)

To confirm that models could be loaded as individual, editable Unity objects (Requirement R8), I assembled the reconstructed Winchester Cathedral cluster into a prefab. As shown in Figure 7.2, I grouped the 466 cluster GameObjects (each with separate roof and body meshes; example of smaller models shown in Figure 7.1) under a parent, then created a prefab. Instantiating the prefab demonstrated that each component remained editable and correctly positioned, satisfying the goal of in-engine, user-manipulable output.

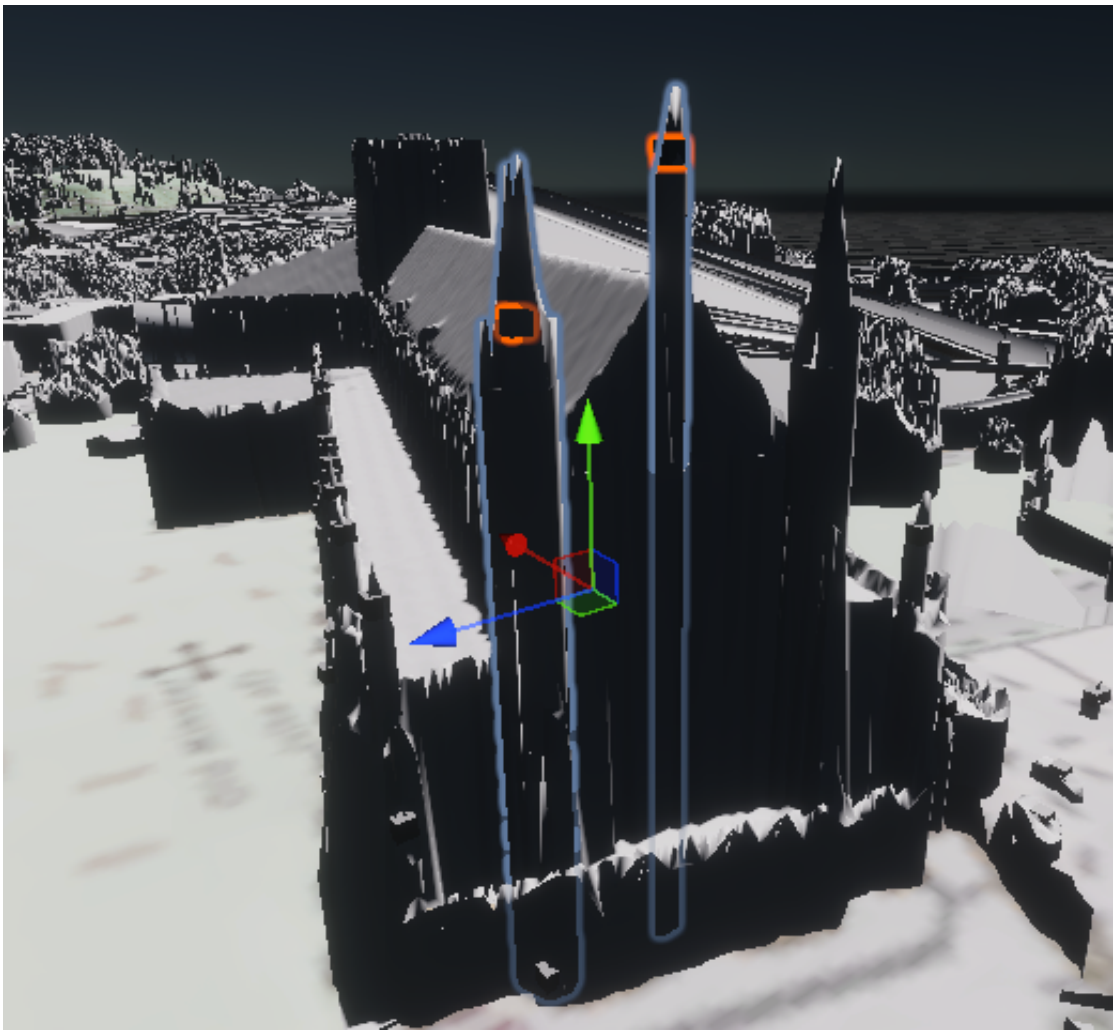

FIGURE 7.1: Example of smaller models built from individual clusters, which often represent architectural details of larger structures.

7.2.2 Mesh Quality Comparison (R7)

Figure 7.3 compares the cuboid-based created at early stage of the development (Step 2, see Chapter 4) with the final mesh-based output (Step 10). The final meshes exhibit smoother surfaces and sharper edges, though some wall polygons show artifacts where the R–table boundary estimation was imprecise. This validates the mesh pipeline (Requirement R7) while highlighting areas for refinement in boundary detection.

7.2.3 Vegetation Classification (R3)

I tested the vegetation flagging (Requirement R3) on the King Alfred roundabout (Figure 7.4). While the pixel-based classifier correctly marked many tree areas in

FIGURE 7.2: Left: instantiated cathedral prefab. Centre: 466 objects generated by the pipeline with their roof and body meshes and placed in the original location of the Winchester Cathedral. Right: grouped cluster objects saved as a prefab asset.

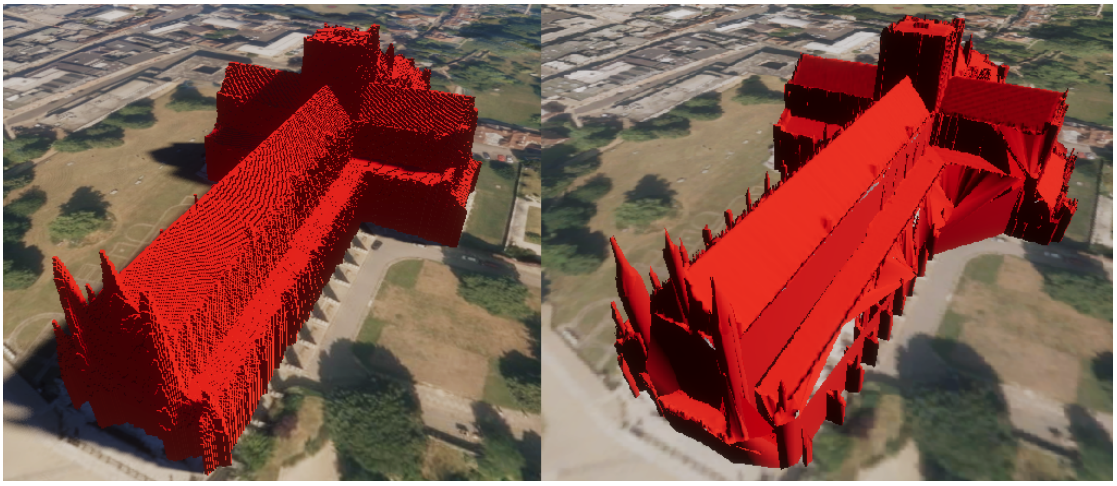

FIGURE 7.3: Left: cuboid-based cathedral. Right: final triangulated and extruded mesh.

green, shadows and non-vegetation green surfaces caused misclassifications. This confirmed the usefulness of the texture approach but also its limitations under uncontrolled lighting.

7.2.4 Cathedral Complex Case Study

This real-world case study evaluates how the clustering and mesh modules perform on a large, architecturally complex structure—the Winchester Cathedral. During model mesh generation (Step 10, Chapter 4), I first recognised an anomalous bridge of triangles connecting the nave to a side building in the full-complex mesh. Examining the body and roof meshes confirmed that boundary estimation was

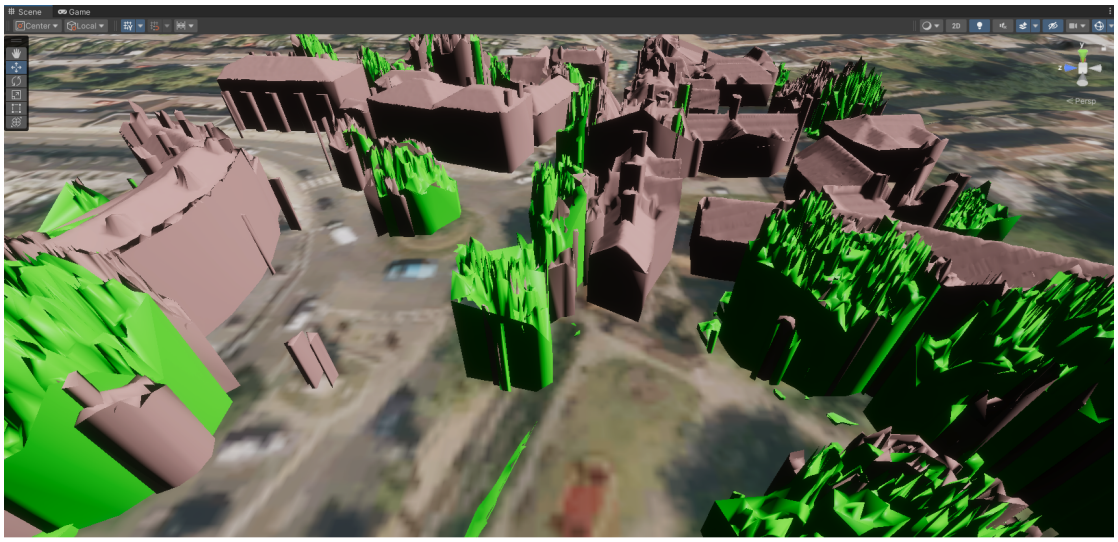

FIGURE 7.4: Vegetation classification results: green = flagged, gray = unflagged.

correct for the walls but that the roof mesh contained spurious triangles and artefacts.

To isolate the cathedral, I ran region-growing clustering on the 634,120-point complex cluster with progressively finer radii:

1. **1.0 m radius, min points = 10,000:** yielded a 90,544-point cathedral cluster.
2. **0.9 m radius:** produced the same 90,544-point cluster (stable).
3. **0.8 m radius:** began fragmenting peripheral points.
4. **0.7 m radius:** split the complex into three subclusters.

Finally, using the 0.7 m threshold, I reconstructed the cathedral from three smaller meshes, which eliminated the anomaly and yielded cleaner roof geometry.

Justification: This case study belongs in the Evaluation chapter because it assesses the pipeline's end-to-end capability—clustering, boundary detection, and mesh generation—on a complex urban landmark, and demonstrates how parameter tuning directly affects output quality.

To avoid cluttering the Evaluation chapter, the full set of images for this case study, including intermediate clustering results and final mesh reconstructions, has been placed in the Appendix (see Appendix D).

7.3 Performance Evaluation

I measured wall-clock times on a 1 km^2 tile (Ryzen 7 3700X, 32 GB RAM):

- **LAZ→CSV (R1):** 120 s
- **CSV→ScriptableObject (R2):** 300 s
- **KD-tree build (R5):** 45 s
- **Clustering (R6):** 20 s
- **Mesh generation (R7):** 10 to 15 s per cluster
- **Terrain creation (R9):** 5 s
- **Unity asset limit and throughput:** Creating and placing 10 000 mesh assets in a single folder took roughly 12 h before Unity reported "creating asset at path failed" due to its 10 000-asset-per-folder limit.
- **Large-scale clustering test:** Running the 1.0 m threshold clustering produced 12 316 clusters (24 632 mesh assets) and required approximately 125 h in total to generate and place all meshes—a single continuous test run that was not monitored moment-to-moment.

Memory peaked at 18 GB during SO Processing and mesh generation. Sectoring and classification filtering (Requirements R2, R4) successfully prevented out-of-memory crashes.

7.4 External Comparison

7.4.1 Visual Comparison with Google Earth: LOD Potential

At the beginning of the project, one of the most encouraging early results came from a very simple setup: using clustered point cloud data to generate a Unity terrain object, and layering it with a basic aerial texture. This minimal configuration—achieved with relatively little effort—already produced a recognisable

representation of Winchester Cathedral when viewed from a bird's-eye perspective. As shown in Figures 7.5 and 7.6, the visual similarity to the Google Earth view is surprisingly close, considering the low detail and procedural nature of the model.

FIGURE 7.5: Google Earth view of Winchester Cathedral and surrounding buildings.

While the later stages of the project revealed much greater complexity—especially when working at ground level or with detailed mesh generation—this example is included here for completeness. It demonstrates the potential for generating lightweight, recognisable environments with minimal setup, and highlights a possible application of the pipeline in lower levels of detail (LOD) for exploratory games or flying simulations.

FIGURE 7.6: Unity view of the same area generated from clustered point cloud data with an aerial texture layer applied.

7.4.2 Urban Reconstruction Quality

Campoverde et al. [4] presented an LOD2 model of Oude Markt with crisp wall geometry and robust vegetation occlusion handling (Figure 7.7). Compared to my output (Figure 7.4), their wall definitions were cleaner and vegetation removal more reliable, though my roof detail was comparable. This underscores the need for more advanced occlusion filtering and wall-detection methods.

7.4.3 Building Separation

Huang et al. [14] achieved clear per-building meshes using planar segmentation (Figures 7.9, 7.10). My approach (Figure 7.8) sometimes merged adjacent buildings or fragmented single structures. Incorporating plane detection or edge classification would likely improve cluster separation (Requirement R6).

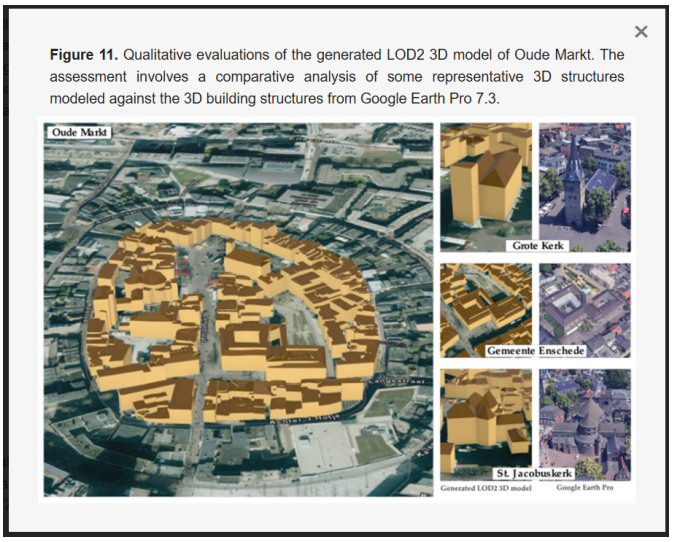

FIGURE 7.7: LOD2 3D reconstruction of Oude Markt (Campoverde et al., 2024).

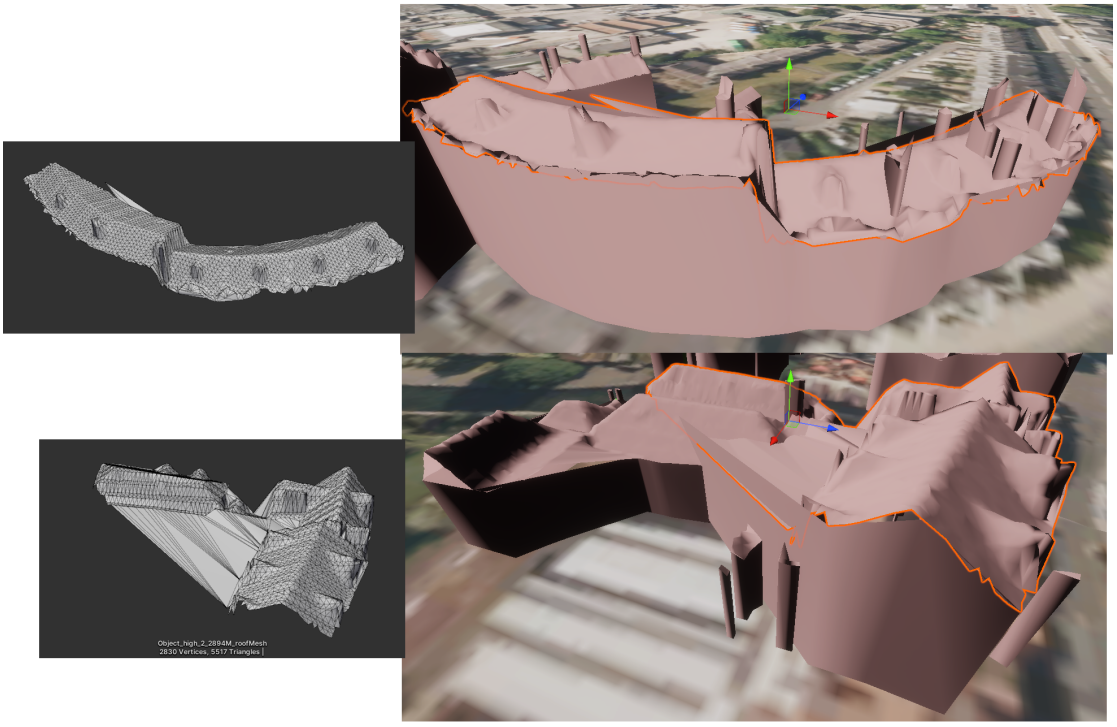

FIGURE 7.8: My building models showing occasional merging or fragmentation.

7.5 Limitations and Lessons Learned

Evaluating the prototype posed its own challenges. As an Editor-only Unity tool rather than a standalone application, user-group testing was not feasible—there was no distributable build. Published benchmarks for raw LiDAR import in this context are virtually non-existent.

I therefore measured success against my original goals (G1–G4):

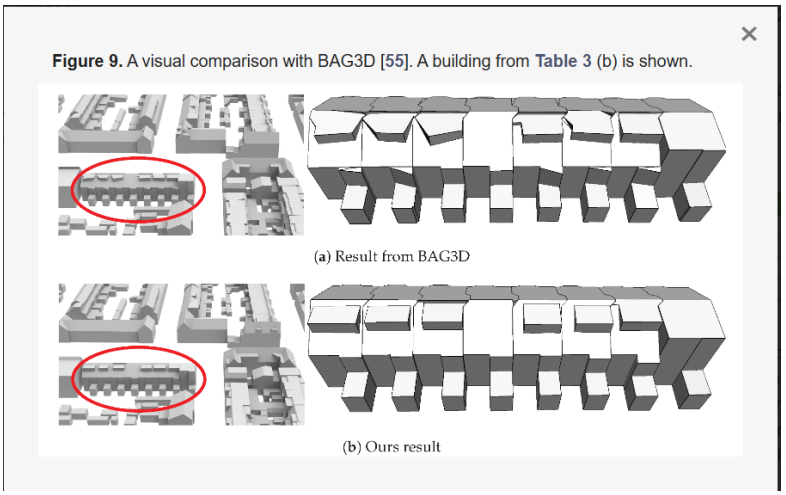

FIGURE 7.9: Building separation results (Huang et al., 2022).

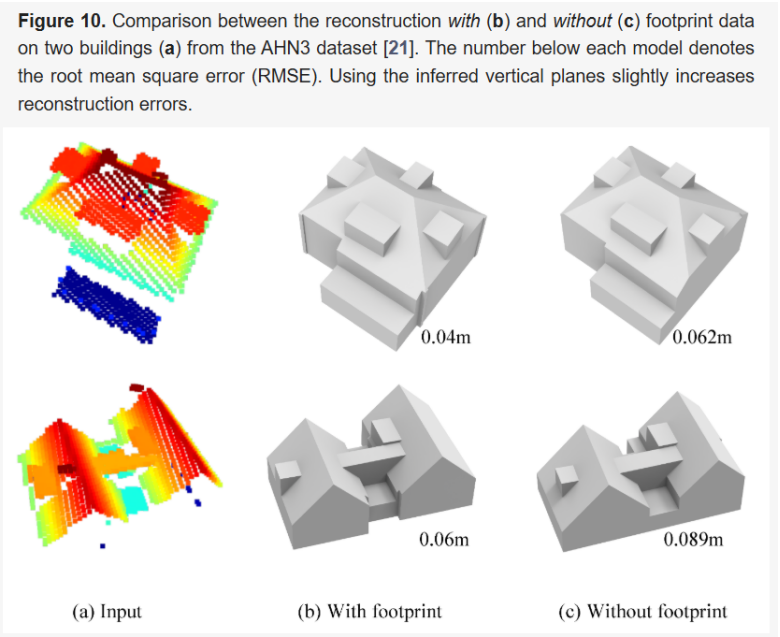

FIGURE 7.10: Planar roof reconstruction (Huang et al., 2022).

- **G1:** LiDAR and DTM import and meshing—achieved.
- **G2:** Point-first processing until meshing—achieved.
- **G3:** Early ground/roof/vegetation classifier—achieved.
- **G4:** Editable Unity objects—achieved.
- **G5:** Faster converters and richer texturing—remains future work.

These are not failures but delimit where this prototype ends. Future evaluation should include:

- Quantitative benchmarks on larger or multi-tile datasets.
- A standalone package or build to enable wider user testing.
- Automated visual QA (e.g. scripted camera paths).

7.6 Method Evaluation

My linear pipeline—from raw LiDAR to in-engine meshes—yielded a functioning prototype but lacked advanced techniques such as planar segmentation, energy-based regularization, or neural-network classification used in related research. A hybrid approach combining point-cloud methods with external GIS or MATLAB preprocessing might better address complex urban scenes.

Nevertheless, the current output serves as a valuable base for manual refinement or integration into game-level workflows, fulfilling the core needs of a rapid, editable urban reconstruction tool.

7.7 Summary

Overall, the prototype met its primary objectives (R1–R10, G1–G4). Performance was acceptable on moderate datasets, and the editor-based workflow proved flexible. Key areas for improvement include conversion speed (G5), vegetation occlusion handling, and per-building separation. This evaluation highlights both the strengths of the implemented pipeline and the path forward for making it production-ready.

Chapter 8

Conclusion

8.1 Achievements

I concluded that this prototype met the core aims of the project. I successfully imported airborne LiDAR and a 50 cm DTM into Unity, preserving ground and building elevations. I kept all processing in the point cloud until the final meshing stage, which maximised control over feature extraction. I applied an early ground/roof/vegetation classifier to reduce data volume and focus on relevant objects, and I generated editable Unity prefabs for each building cluster, enabling first-person exploration and manual refinement. Performance on a single 1×1 km tile was acceptable: CSV and DTM serialisation, classification, KD-tree building and clustering each completed in under 15 minutes, individual cluster meshes were generated in seconds, and overlaying topographic and aerial imagery onto the generated terrain confirmed good alignment with real-world features.

8.2 Limitations

Despite these successes, several limitations became clear. I was only able to process one tile at a time, which made multi-tile processing impractical without better memory management. The workflow was limited to the Unity Editor, so I could not create a standalone build for external testing. The basic classification method sometimes merged adjacent buildings or fragmented single ones, and texturing relied on static overlays without using procedural materials or dynamic occlusion

handling. Quality assurance remained manual, based on visual inspection. Processing a full 1×1 km tile on a regular PC often took several days, which was still acceptable compared to the time needed for manual modelling, but showed the need for faster solutions.

Another important limitation was the choice of Unity as the main environment for data processing. I spent significant effort managing memory and handling import issues rather than focusing on the main algorithms. If I were to repeat this work, I would handle the data preparation and classification externally—using MATLAB or QGIS—and import the ready-made prefabs into Unity instead of building them within the Editor. Finally, working on this project alone proved overly demanding. For a task of this scale and complexity, I would strongly recommend assembling a capable, multidisciplinary team from the start.

8.3 Future Work

If I were to continue this project, the first priority would be to improve the foundations. This would involve shifting the core data preparation—such as filtering, classification, and noise removal—into MATLAB or a GIS environment like QGIS. This approach would make the pipeline more reliable and avoid many of the limitations I encountered with Unity's Editor. I would also aim to build a strong development team to share responsibility, ensure wider expertise, and support the technical and design challenges involved.

Once the core workflow was stabilised, I would work on improving building separation using planar roof detection and deep-learning edge classifiers. Classification and texturing could be enhanced by applying modern semantic segmentation models, biome-aware terrain texturing, and procedural vegetation placement. These additions would improve the realism and usability of the generated environment.

I also planned to explore more advanced computer vision methods. These would include expanding the range of classifier architectures, applying noise-reduction and super-resolution techniques to both LiDAR and aerial imagery, and using edge extraction from textures to better define building outlines. These improvements could help bring the visual output closer to reality, which had always been one of my goals.

I intended to automate the quality assurance process by scripting camera paths and using image-based metrics to detect alignment errors and mesh artefacts without

relying solely on manual checks. In parallel, I planned to run a small user study involving game designers to gather feedback and refine the overall workflow and usability of the system.

With a more stable and robust pipeline in place, I would then return to the challenge of scaling the system. This would include implementing on-the-fly tiling, level-of-detail (LOD) streaming, and parallelised LAZ-to-octree conversion to support larger datasets. I also planned to deploy the pipeline on Iridis HPC using a Linux version of Unity to take advantage of GPU acceleration, reduce processing times, and allow larger-scale tests that could identify and resolve structural issues earlier in the process.

Finally, my long-term goal was to develop a playable game prototype. This would allow users to explore a realistic reconstruction of Winchester in first person and, one day, step inside a detailed virtual model of my favourite cathedral—bringing together geospatial accuracy and immersive interactivity.

Appendix A

Archive

All files required to run the prototype—apart from the Unity scene itself—are included in archive.zip. The archive contains the full folder structure and all relevant C# scripts, prefabs, terrain files, point cloud data, and configuration assets used during development. This package represents the complete environment needed to reproduce or extend the pipeline as implemented, assuming a compatible Unity setup.

Archive ZIP Contents

All files required to run the prototype, excluding the Unity scene, are included in `archive.zip`. The table below summarises the folder structure and contents.

Folder	**Contents**
archive.zip	Archive/
Archive/	Assets/, LazPointCloudReconstruction.sln
Archive/Assets/	Data/, Editor/, Materials/, Mesh/, PDAL/, Prefabs/, Scripts/, Triangle/
Archive/Assets/Data/	Assets/, Clusters/, MClusters/, Scripts/, aerial.terrainlayer, infoMap.terrainlayer, su4829.laz, su4829.png, su4829_DTM_50CM.asc, su4829_l2_250_03flip.jpg

Archive/Assets/Data/Scripts/	BasePointCollection.cs, ClassifierCollection.cs, ClusterCollectionListAsset.cs, ClusterData.cs, ContainerData.cs, DatabaseData.cs, FaceData.cs, KDTree.cs, KDTreeCollection.cs, MeshData.cs, PointData.cs, PointDataCollection.cs, PointEntry.cs
Archive/Assets/Editor/	ASCIIGridConverter.cs, BasePointCollectionEditor.cs, ClassifierCollector.cs, ClusterDivider.cs, ClusterMeshProcessor.cs, CSVProcessor.cs, Delaunay2D.cs, GameManagerEditor.cs, JsonFactory.cs, KDTreeCollectionCreator.cs, KDTreeCollectionEditor.cs, ModelMeshFactory.cs, OptimisedPrefabFactory.cs, PDALProcessor.cs, PrefabFactory.cs, SOProcessor.cs, TerrainFactory.cs, VegetationDetector.cs, VegetationTextureExtractor.cs
Archive/Assets/Editor/EditMode/	WhiteBoxTests.cs
Archive/Assets/Prefabs/	CathedralHighPrefab.prefab
Archive/Assets/Scripts/	CameraController.cs, ClusterPrefabProcessor.cs, GameEditorManager.cs, GameManager.cs, Init.cs
Archive/Assets/Materials/	(placeholder)
Archive/Assets/Mesh/	(placeholder)
Archive/Assets/PDAL/	(Library)
Archive/Assets/Triangle/	(Library)

Appendix B

Code Documentation

B.1 Editor Tools

B.1.1 GameManagerEditor.cs

Class Overview:
The `GameManagerEditor` is a custom Unity inspector designed to manage the full data processing and visualisation pipeline through a GUI interface. It provides grouped inspector controls for converting data, managing assets, filtering classifiers, extracting clusters, generating terrain, calculating meshes, and creating prefabs.

Core Responsibilities:

- Launches PDAL conversion from LAZ to CSV.
- Runs ScriptableObject creation with optional DTM and vegetation inputs.
- Provides UI for KD-tree, clustering, mesh processing and prefab generation.
- Supports asset-based and manual cluster selection.
- Offers test utilities for terrain and KDTree building.
- Computes terrain height statistics from the selected point collection.

Hardcoded and Fixed Assumptions:

- Output folders are fixed (e.g., `Assets/Data/Assets`, `Assets/Data/MClusters`).
- Grid divider values are predefined (0, 4, 8, 16, 64).
- Uses editor-only classes and methods (e.g., `AssetDatabase`, `EditorGUILayout`).

Unity-Specific Behaviour:

- Requires UnityEditor and uses `CustomEditor` for layout.
- Extensively uses `GUILayout` and `EditorGUILayout` for grouped UI.
- Utilises serialized properties and toggles for runtime-exposed pipeline parameters.
- Computes minimum, maximum, and vertical span of terrain heights using either raw or DTM-adjusted values.

Dependencies:

- **Unity:** `UnityEngine`, `UnityEditor`, `EditorGUILayout`, `GUILayout`, `Debug`
- **Internal:** `GameManager`, `SOProcessor`, `VegetationTextureExtractor`, `ClassifierCollector`, `KDTreeCollectionCreator`, `ClusterDivider`, `ClusterMeshProce` `ModelMeshFactory`, `TerrainFactory`

B.1.2 PDALProcessor.cs

Class Overview: The `PDALProcessor` is a static utility class responsible for converting LAZ files to CSV format using an embedded PDAL executable. It executes an external process (`pdal.exe`) with appropriate arguments and handles logging, timeout, and output refresh via Unity's `AssetDatabase`.

Core Method:

- `ConvertLAZtoCSV(...)` — Runs the PDAL `translate` command using absolute file paths for both input LAZ and output CSV. Captures and logs PDAL's standard output and error streams.

Hardcoded and Fixed Assumptions:

- `DefaultTimeoutMs` is set to **5 minutes** (300,000 ms), which may need to be increased for large LAZ files or slower machines.
- Uses a fixed PDAL command structure: `translate input.laz output.csv`. Other PDAL pipeline features (e.g., filters, reprojection) are not supported in this wrapper.
- Assumes that `pdal.exe` and required DLLs are placed under a single directory (`Assets/Plugins`), and that this folder is correctly passed as the working directory.
- Only LAZ → CSV conversion is supported — there's no reverse operation or extended PDAL pipeline integration.

Unity-Specific Behaviour:

- Automatically triggers `AssetDatabase.Refresh()` after successful conversion, so Unity can immediately detect the new CSV asset.

Dependencies:

- **System:** `System.Diagnostics`, `System.IO`, `System` (for `Exception`, `Path`, `Process`, etc.)
- **Unity:** `UnityEngine.Debug`, `UnityEditor.AssetDatabase`
- **Internal:** `None` (does not depend on other custom project classes)

B.1.3 SOProcessor.cs

Class Overview: The `SOProcessor` class is a static Unity utility used for converting CSV and ASCII DTM input files into Unity-compatible `ScriptableObject` assets, specifically `BasePointCollection`. It supports cyclic rotation, DTM ground height integration, optional vegetation detection using texture masks, and spatial subdivision of large datasets into square sectors. It includes two main entry points: one for converting ASCII DTM files, and another for converting CSV + DTM (+ vegetation).

Core Methods:

- `ConvertCSVToScriptableObject(...)` — Main entry point for serialising a CSV file into a BasePointCollection asset, optionally integrating DTM and vegetation texture, and splitting into sectors.
- `ConvertDTMToScriptableObject(...)` — Extracts point data from an ASCII DTM grid and saves it as a BasePointCollection asset.
- `DivideCollectionIntoSectors(...)` — Divides a BasePointCollection into equal-sized square sectors and saves each as a separate asset.

Helper Methods:

- `ApplyCyclicRotation(...)` — Reorders coordinates in a (Y, Z, X) cyclic permutation.
- `SaveDtmBasePointCollection(...)` — Internal method for building a BasePointCollection from raw DTM data.

Hardcoded and Fixed Assumptions:

- Uses a fixed cyclic permutation: $(x, y, z) \rightarrow (y, z, x)$ to align with Unity axis conventions.
- CSV input expects at least 9 columns, and fixed indices are used: x=0, y=1, z=2, classification=8.
- A hardcoded `scaleFactor` of **1.0f** is applied when parsing CSV — this may need to change for different input scales.
- Pixel size for vegetation classifier is fixed at **1000 / textureWidth**, assuming a 1 km² image.
- Grid division for sectoring is square only (i.e., `gridCellsPerSide × gridCellsPerSide`).
- The DTM-to-point mapping assumes grid corner alignment without half-cell offsets.

Unity-Specific Behaviour:

- Asset creation and folder paths are managed using `AssetDatabase.CreateAsset`, `SaveAssets`, and `Refresh`.

- Logging and debug outputs are printed using `Debug.Log()`.
- The class is entirely dependent on Unity's Editor runtime.

Dependencies:

- **System:** `System`, `System.IO`, `System.Globalization`, `System.Collections.Generic`, `Math`
- **Unity:** `UnityEngine`, `UnityEditor`, `Debug`, `Texture2D`, `Random`
- **Internal:**
  - `PointData` (data structure)
  - `BasePointCollection` (ScriptableObject)
  - `VegetationDetector.ReadDTM(...)` — to parse ASCII DTM
  - `VegetationTextureExtractor.ExtractIsVegetation(...)` — for generating boolean vegetation mask

B.1.4 ClassifierCollector.cs

Class Overview:
The `ClassifierCollector` class provides editor-only functionality to filter and export subsets of point cloud data based on LAS classification values. It supports additional filtering using vegetation flags.

Core Method:

- `ExtractClassifierCollection(...)` — Filters points by classification and vegetation, and saves a new `ClassifierCollection` asset with metadata and expected point count.

Hardcoded and Fixed Assumptions:

- A hard-coded dictionary stores expected counts and filenames for known classifiers.
- Output folder must already exist and be Unity asset-compatible.

- Names are generated dynamically depending on classifier and vegetation filter combination.

Unity-Specific Behaviour:

- Depends on `AssetDatabase` to save and register ScriptableObject assets.
- Uses `Debug.Log()` for output and error reporting.

Dependencies:

- **System:** `System.Collections.Generic`, `System.Globalization`, `System.IO`, `System.Linq`
- **Unity:** `UnityEngine`, `UnityEditor`
- **Internal:** `BasePointCollection`, `ClassifierCollection`, `PointData`, `VegetationFilte` enum

B.1.5 VegetationDetector.cs

Class Overview:
The `VegetationDetector` is a static editor utility used to load DTM raster files and perform spike-based vegetation detection for internal testing purposes.

Core Methods:

- `ReadDTM(...)` — Loads an ESRI ASCII raster file and extracts metadata (grid size, origin, cell size, nodata) and height values.
- `DetectVegetationCandidates(...)` — Identifies cells that stand out as height spikes compared to neighbors.

Hardcoded and Fixed Assumptions:

- Cell size and nodata value are initially set to 0 and overridden from file.
- Floating-point NaN is used for nodata.
- Intended for editor-time terrain and vegetation experiments.

Unity-Specific Behaviour:

- Uses `Debug.Log()` and returns results suitable for ScriptableObject workflows.

B.1.6 VegetationTextureExtractor.cs

Class Overview:
The `VegetationTextureExtractor` class provides editor-only utilities to classify and debug vegetation data in texture maps. It's mainly used for visualising and filtering vegetation areas before asset generation.

Core Methods:

- `ExtractIsVegetation(...)` — Determines whether each pixel in a texture is likely to represent vegetation.
- `TestVegetationTexture(...)` — Instantiates coloured cubes to visualise vegetation and background.
- `TestTextureStatistics(...)` — Logs RGB stats for corner pixels and global min/avg/max for tuning.

Hardcoded and Fixed Assumptions:

- Green channel dominance is used as the main detection strategy.
- Visual test cubes are placed at fixed height (5.0f) and positioned using pixel indices.
- Commented code hints at earlier plans to output KDTree assets.

Unity-Specific Behaviour:

- Requires `Texture2D.GetPixels()` and other editor-only APIs.
- Relies on runtime types but is not safe for runtime builds without guards.

Dependencies:

- **System:** `System.Collections.Generic`, `System.Globalization`, `System.IO`
- **Unity:** `UnityEngine`, `UnityEditor` (optional save logic)

B.1.7 KDTreeCollectionCreator.cs

Class Overview:
The `KDTreeCollectionCreator` is an editor-only helper class designed to convert any `BasePointCollection` into a `KDTreeCollection` asset by calculating nearest neighbours and storing the data for later use in clustering or mesh building.

Core Method:

- `CreateKDTreeCollection(...)` — Copies point data from a `BasePointCollection`, calculates a KD-tree for neighbour lookup, and saves it as a `KDTreeCollection` asset. Logs sample neighbour data and a random point.

Hardcoded and Fixed Assumptions:

- Output asset name is auto-generated based on input asset's name, suffixed with `KDTree.asset`.
- Uses `ScriptableObject.CreateInstance` and saves directly to the output folder.
- Mesh processing is explicitly not triggered, allowing deferred control.

Unity-Specific Behaviour:

- Depends on `AssetDatabase` and is intended for use only in the Unity Editor.
- Logs key output to `Debug.Log()`, including sampled neighbour and random point info.

Dependencies:

- **System:** `System.Collections.Generic`, `System.Linq`, `System.IO`
- **Unity:** `UnityEngine`, `UnityEditor`
- **Internal:** `BasePointCollection`, `KDTreeCollection`, `PointData`

B.1.8 KDTreeCollection.cs

Class Overview: The `KDTreeCollection` is a serialised asset derived from `BasePointCollection` that stores spatial indexing and mesh-related data used for clustering and geometry reconstruction. In earlier stages of the project, it was used purely as a data container; however, it was later extended with functionality to compute and log neighbour data.

Core Methods:

- `ComputeNeighbors(...)` — Builds an internal KD-tree and assigns nearest neighbours for each point. Supports 3D and horizontal-only modes.
- `LogSampleNeighborData(...)` — Prints a sample of neighbour indices and squared distances from the internal Neighbors dictionary.

Fields and Responsibilities:

- `Neighbors` — A dictionary mapping each point's index to its nearest neighbours and corresponding squared distances.
- `boundaryPoints` — Optional precomputed boundary (R-table) points used in later mesh generation.
- `DominantGradient`, `RefinedCenter`, `EdgePoints` — Fields related to geometry analysis and cluster refinement.
- `mesh`, `polyhedronMesh` — Mesh data containers produced during roof and body generation.
- `center` — Precomputed cluster centre in world coordinates.

Hardcoded and Fixed Assumptions:

- Neighbour data is recomputed on demand and not cached across sessions.
- Horizontal-only mode sets y-values to zero before KD-tree construction.
- Neighbour IDs are mapped via point ID, not array index.

Unity-Specific Behaviour:

- Designed as a `ScriptableObject` that is saved and reloaded as an asset in Unity.
- Uses `Debug.Log()` to report progress and data samples during development.

Dependencies:

- **System:** `System`, `System.Collections.Generic`, `System.Linq`
- **Unity:** `UnityEngine`
- **Internal:** `BasePointCollection`, `PointData`, `KDTree`, `PointEntry`, `MeshData`

B.1.9 KDTree.cs

Class Overview:
The `KDTree` class implements a spatial indexing structure using a 3D KD-tree. It enables fast k-nearest neighbour (k-NN) queries in 3D space and is used throughout the pipeline to analyse spatial proximity between points. The tree is built recursively and stores an indexed hierarchy of points using a depth-dependent axis split (x, y, z).

Core Method:

- `GetKNearestNeighborsWithIndex(...)` — Returns a list of the k nearest neighbours to a given query point, as (index, squared distance) pairs. Used by `KDTreeCollection` to precompute neighbour relationships.

Internal Methods and Logic:

- `BuildTree(...)` — Recursively splits points to construct the balanced tree.
- `Search(...)` — Performs recursive neighbour search using backtracking and pruning based on axis-aligned distance.
- `SquaredDistance(...)` — Calculates squared Euclidean distance between a point and a target coordinate.
- `ComparePoints(...)` — Determines the sort order during construction by comparing axis-specific values.

Hardcoded and Fixed Assumptions:

- Operates in exactly three dimensions (x, y, z).
- Point identity is derived from their index in the input list.
- Neighbour distances are calculated as squared values for efficiency.

Unity-Specific Behaviour:

- The class is engine-agnostic and does not use Unity APIs, but takes `Vector3` as query input for interoperability.
- Intended to be wrapped by Unity-specific logic such as `KDTreeCollection`.

Dependencies:

- **System:** `System.Collections.Generic`, `System.Linq`
- **Unity:** `UnityEngine.Vector3`
- **Internal:** `PointData` (used for spatial input)

B.1.10 ClusterDivider.cs

Class Overview:
The `ClusterDivider` class provides region-growing clustering logic that groups nearby points in a `KDTreeCollection` based on horizontal proximity. Clusters are saved as separate assets and optionally aggregated into a list asset. It is primarily used to prepare data for mesh generation or spatial filtering.

Core Method:

- `DivideCollection(...)` — Main method that segments a KDTreeCollection into clusters based on a horizontal distance threshold. Applies point count filters, saves individual clusters, and optionally generates a master list asset.

Hardcoded and Fixed Assumptions:

- Clustering uses region-growing with squared distance on the x–z plane (ignoring y).
- Cluster names follow the format: `Prefix_Index_Count` where prefix is extracted from input asset name (e.g. `Veg_low`).
- Prefix detection uses regex patterns tied to expected naming schemes like `All_low`, `Object_high`, etc.
- Neighbour relationships must already be computed and included in the input asset.

Unity-Specific Behaviour:

- Clusters and list assets are saved using `AssetDatabase.CreateAsset()` and refreshed via `AssetDatabase.Refresh()`.
- Relies on UnityEditor functionality and is disabled in runtime builds.

Dependencies:

- **System:** `System.Collections.Generic`, `System.IO`, `System.Linq`, `System.Text.RegularExpressions`
- **Unity:** `UnityEngine`, `UnityEditor`
- **Internal:** `KDTreeCollection`, `PointData`, `ClusterCollectionListAsset`

B.1.11 ClusterMeshProcessor.cs

Class Overview:
The `ClusterMeshProcessor` class processes `KDTreeCollection` clusters to generate 3D mesh data, including roofs and side walls (polyhedra), based on point cloud geometry and DTM values. It includes custom triangulation and polygon filtering logic, and supports saving processed assets as new ScriptableObjects.

Core Responsibilities:

- Converts cluster points into an internal `PointEntry` format, applying cyclic coordinate reordering.

- Extracts ordered boundary points using a dynamic angular binning approach (R-table).
- Triangulates roof geometry using Triangle.NET and filters faces based on 2D polygon inclusion.
- Constructs side wall (body) geometry using local-space projection and DTM ground height.
- Wraps the resulting geometry into `MeshData` and saves results if requested.

Hardcoded and Fixed Assumptions:

- Coordinate rotation is toggled globally via a boolean `useSwap`.
- Boundary resolution (number of bins) ranges from 9 to 720 depending on point count.
- Roof triangulation uses Triangle.NET; centroid inclusion test is ray-casting in XZ plane.
- Processed cluster names are suffixed with "M", and output list asset uses "_MList".

Unity-Specific Behaviour:

- Uses `ScriptableObject.CreateInstance`, `AssetDatabase.CreateAsset`, and `AssetDatabase.SaveAssets`.
- Uses `Debug.Log` for error reporting and runtime status updates.

Dependencies:

- **System:**
  `System`, `System.Collections.Generic`, `System.Linq`, `System.IO`, `System.Text.RegularExpressions`
- **Unity:**
  `UnityEngine`, `UnityEditor`
- **Third-Party:**
  `TriangleNet`, `TriangleNet.Geometry`, `TriangleNet.Meshing`, `TriangleNet.Topology`

- **Internal:**
  `PointData`, `KDTreeCollection`, `KDTree`, `ClusterData`, `MeshData`, `FaceData`, `PointEntry`, `ClusterCollectionListAsset`

B.1.12 ModelMeshFactory.cs

Class Overview:
The `ModelMeshFactory` class creates GameObjects and mesh assets from processed `KDTreeCollection` clusters. It generates visual representations in the Unity scene by constructing both roof and body geometry, saving assets to organised folders, and applying optional mesh filtering and winding rules.

Core Responsibilities:

- Generates scene GameObjects for each cluster with mesh and metadata components.
- Builds roof and body meshes from cluster-provided `MeshData`.
- Saves mesh assets to structured output directories grouped by batch.
- Applies optional edge-length filtering and winding order reversal during mesh creation.

Hardcoded and Fixed Assumptions:

- Default material is saved as `DefaultMaterial.asset` in the material output folder.
- Each cluster is assigned to a group folder (e.g., `ClusterGroup_0`) to prevent overcrowding.
- Filtering threshold for triangle edges is fixed at **2.0 units**.
- If no prefab is given, an empty GameObject is created.

Unity-Specific Behaviour:

- Uses `AssetDatabase` for saving meshes and refreshing the project view.
- Relies on `MeshFilter` and `MeshRenderer` to attach generated meshes.

- Creates a new GameObject for each cluster and embeds child roof/body meshes under it.

Dependencies:

- **System:**
  `System.IO`, `System.Collections.Generic`, `System`
- **Unity:**
  `UnityEngine`, `UnityEditor`
- **Internal:**
  `KDTreeCollection`, `PointData`, `MeshData`, `FaceData`, `PointEntry`

B.1.13 TerrainFactory.cs

Class Overview:
The `TerrainFactory` class is a static utility used to generate Unity terrain objects from processed point collections. It supports both standard and DTM-based terrain height values. It bins the data into a grid, normalises the heights, and produces a `TerrainData` asset, which is then placed in the Unity scene with optional vertical offset to place ground level at Y=0.

Core Methods:

- `CreateTerrainFromCollection(...)`
  — Computes bounds, bins point data into a height grid, normalises elevation, and returns a configured `TerrainData` object.
- `CreateTerrainGameObject(...)`
  — Instantiates a terrain GameObject in the scene, sets its name, applies vertical offset based on minimum height.

Hardcoded and Fixed Assumptions:

- Height normalisation range is `[minY, maxY]` and clamped to a minimum span of `1.0` to avoid flat output.
- The terrain cell grid is square, evenly subdivided by the given resolution.

- The terrain always aligns its bottom to Y=0 by lifting by `-minY`.

Unity-Specific Behaviour:

- Uses `Terrain.CreateTerrainGameObject()` to instantiate terrain from `TerrainData`.
- Applies positional offset to reposition terrain such that the ground aligns with Y=0.

Dependencies:

- **System:**
  `System`,
  `System.Collections.Generic`,
  `System.IO`
- **Unity:**
  `UnityEngine`,
  `UnityEditor`
- **Internal:**
  `BasePointCollection`, `PointData`

B.2 Runtime Scripts

B.2.1 CameraController.cs

Class Overview:
A simple free-look first-person camera controller for moving and rotating the camera using keyboard and mouse input.

Controls:

- `WASD` — Move forward, back, and strafe.
- `Mouse Movement` — Adjust camera pitch and yaw.

Parameters:

- `speed` — Movement speed.
- `sensitivity` — Mouse look sensitivity.

B.2.2 Init.cs

Class Overview:
Initialises the player GameObject, adds a character controller and camera, and enables full first-person controls including gravity and jumping.

Core Responsibilities:

- Creates and positions player and camera objects.
- Handles basic movement, camera pitch/yaw, and jumping.
- Locks and hides the mouse cursor for immersive first-person control.

Controls:

- `WASD` — Move forward/back/strafe.
- `Mouse Movement` — Pitch and rotate view.
- `Space` — Jump.
- `Q / E / Z / X` — Rotate and pitch camera manually.
- `Ctrl + Q` — Exit Play Mode.

Internal Values:

- `CharacterController` component used for movement.
- `gravityValue`, `jumpHeight`, and `pitchRange` govern motion.
- Automatically places player at a safe starting height.

B.2.3 GameManager.cs

Class Overview:
Acts purely as a container for all serialised project fields used in the editor pipeline. This class has no logic of its own.

Exposed Parameters:

- **Paths:** `csvFilePath`, `lazFilePath`, `pdalPath`, `dataPath`
- **Point Collections:** `inputPointCollection`, `terrainCollection`, `kdTreeCollectionIn` `clusteringCollectionInput`, etc.
- **Thresholds and Masks:** `expectedPointCount`, `classifierMask`, `clusterThreshold`, etc.
- **Asset Management:** `saveClusters`, `saveMClusters`, `clusterListAsset`, etc.
- **Vegetation:** `vegetationTexture`, `gridDivider`, `vegetationFilter`
- **ASCII and DTM:** `asciiDTMFile`, `asciiAssetOutput`, `doRotation`, `isAsciiGrid`, etc.
- **Visual Settings:** `terrainResolution`, `terrainHeight`, `minHeight`, `maxHeight`, `heightOffset`

Appendix C

Testing Details

C.1 White Box Tests

I systematically exercised each module in the point-cloud processing pipeline with a suite of white-box tests, verifying both normal operation and edge-case behaviour. Table C.1 lists every test by its Use-Case ID (UC110–UC730) along with a brief description of its intent. In summary, I covered:

- **LAZ → CSV Conversion (UC110–UC122):** I verified that the source LAZ file was present, simulated PDAL translation via a stub, checked the CSV schema (header and data rows), and confirmed record counts against the LAS report.
- **CSV → ScriptableObject Serialization (UC210–UC243):** I tested low-level transforms (cyclic rotation, global rebase), parsed ASCII DTM grids (including off-grid binning), and validated sectoring logic under valid and invalid parameters.
- **Classification Filtering (UC410–UC431):** I exercised null-input and folder-validation cases, extracted single and multiple classifiers (All, Veg, Object), and chained filters to refine subsets.
- **KD-Tree Neighbors Computation (UC510–UC520):** I handled empty point sets, verified k-nearest queries (including a horizontal-only mode), tested sample logging, and confirmed independent behaviour of multiple KD-tree instances.

- **Clustering (UC610–UC630):** I ran region-growing cluster detection with point-count thresholds, tested error conditions for missing or incomplete inputs, and confirmed creation of both individual cluster assets and a cluster-list asset.
- **Mesh Generation (UC710–UC730):** I generated roof and polyhedron meshes for single clusters, batch-processed multiple clusters, and validated saving of mesh assets along with a consolidated mesh-list asset containing metadata.

Each test was implemented as a positive-only NUnit test (with minimal integration points), targeting public APIs or internal helpers via reflection. I used `LogAssert` to verify expected error logs, and standard `Assert` calls to confirm correct return values, asset creation, and data transformations. This appendix therefore provided full traceability from each use-case requirement to its implementation verification in the core pipeline.

Table C.1: Mapping of White-Box Tests to Use-Case IDs

UC Code	**Test Description**
Key	UC110 is unit test of Use Case 1.1
UC110	Check that `su4829.laz` exists in `Assets/Data`
UC111	Convert LAZ to CSV via PDAL stub returns `true`
UC120	Read CSV header and verify it matches "X,Y,Z,Classification"
UC121	Read first data row of CSV and assert all columns parse as numbers
UC122	Verify CSV row count matches the "number of point records" in LAS report
UC210	Apply cyclic rotation $(x, y, z) \to (y, z, x)$ to a `PointData`

Continued on next page

Table C.1 continued from previous page

UC Code	**Test Description**
UC211	Global rebase shifts minimum x, z to zero across a point list
UC212	Combine rotation and rebase transforms correctly
UC220	Verify ASCII DTM file `su4829_DTM_50CM.asc` exists in `Assets/Data`
UC221	Read `ncols` and `nrows` from DTM header and assert positive
UC222	Parse first DTM data value as a valid float
UC230	Point at (0.5,0.5) bins to cell [1,0] and assigns $dtmY = 30$
UC231	Off-grid random offsets on a 3×3 grid bin to correct DTM cell values
UC232	Random offsets (cell size = 0.5) on a 2×2 DTM produce correct bins
UC240	1×1 sectoring yields one "_section_" asset containing all points
UC241	2×2 sectoring yields four assets whose points sum to the original count
UC242	Invalid grid size (0) logs error and creates no sector assets
UC243	Null input to sector division logs error without throwing exception
UC410	Null classifier input logs error, produces no asset
UC411	Invalid classifier output folder logs error, produces no asset

Continued on next page

Table C.1 continued from previous page

UC Code	**Test Description**
UC412	Single-classifier "All" filter includes both veg and non-veg points
UC413	Multi-classifier "All" sums expected counts and names asset correctly
UC420	Veg filter includes only vegetation points
UC421	Object filter includes only non-vegetation points
UC431	Chained filtering: "All" then "Object" yields correct subset
UC510	No points logs error, leaves `Neighbors` null
UC511	`ComputeNeighbors` populates `Neighbors` with correct counts
UC512	`horizontalOnly` ignores vertical differences in neighbor lookup
UC513	`LogSampleNeighborData` before compute logs warning
UC514	`LogSampleNeighborData` after compute logs sample entries
UC520	Multiple KD-Tree instances maintain independent neighbor sets
UC610	Null KDTreeCollection input to clustering logs error, returns null
UC611	Clustering without neighbors logs error, returns null
UC612	Clustering logs counts and returns accepted clusters with correct names
UC620	Saving clusters as assets creates one asset per cluster

Continued on next page

Table C.1 continued from previous page

UC Code	**Test Description**
UC630	Saving list of clusters creates one consolidated list asset
UC710	Compute roof & polyhedron mesh for a single cluster
UC720	Batch mesh generation; processed names suffixed with "M"
UC730	Saving processed mesh assets and mesh-list asset with metadata

Appendix D

Cathedral Gallery

This appendix presents the full visual record of the Cathedral Complex case study, showing the region-growing clustering process, intermediate mesh results, and final reconstructed model.

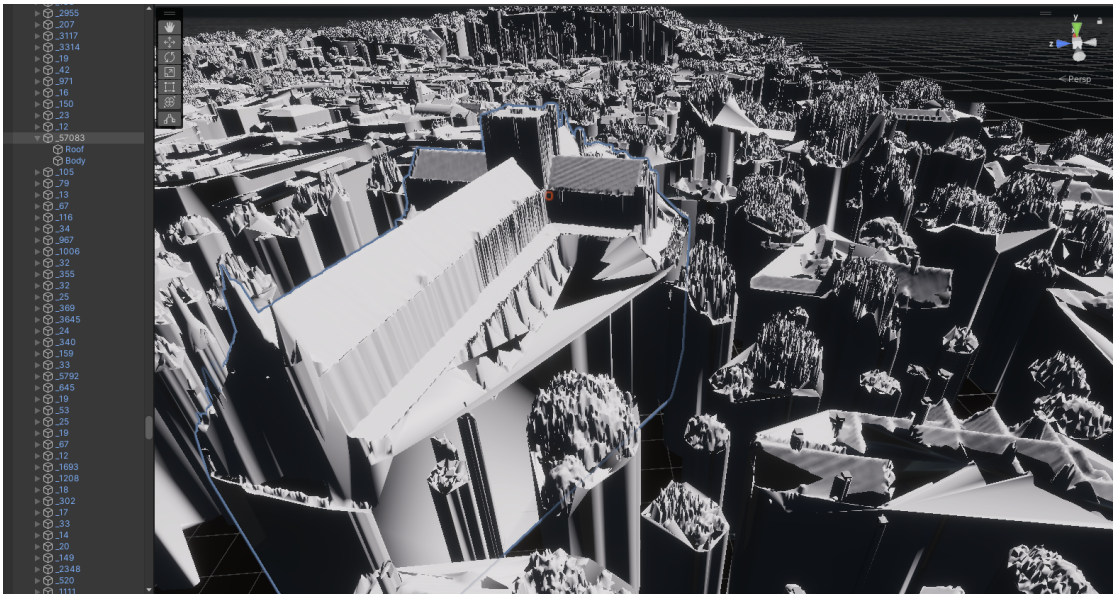

FIGURE D.1: Structural anomaly identified during large cluster list model generation.

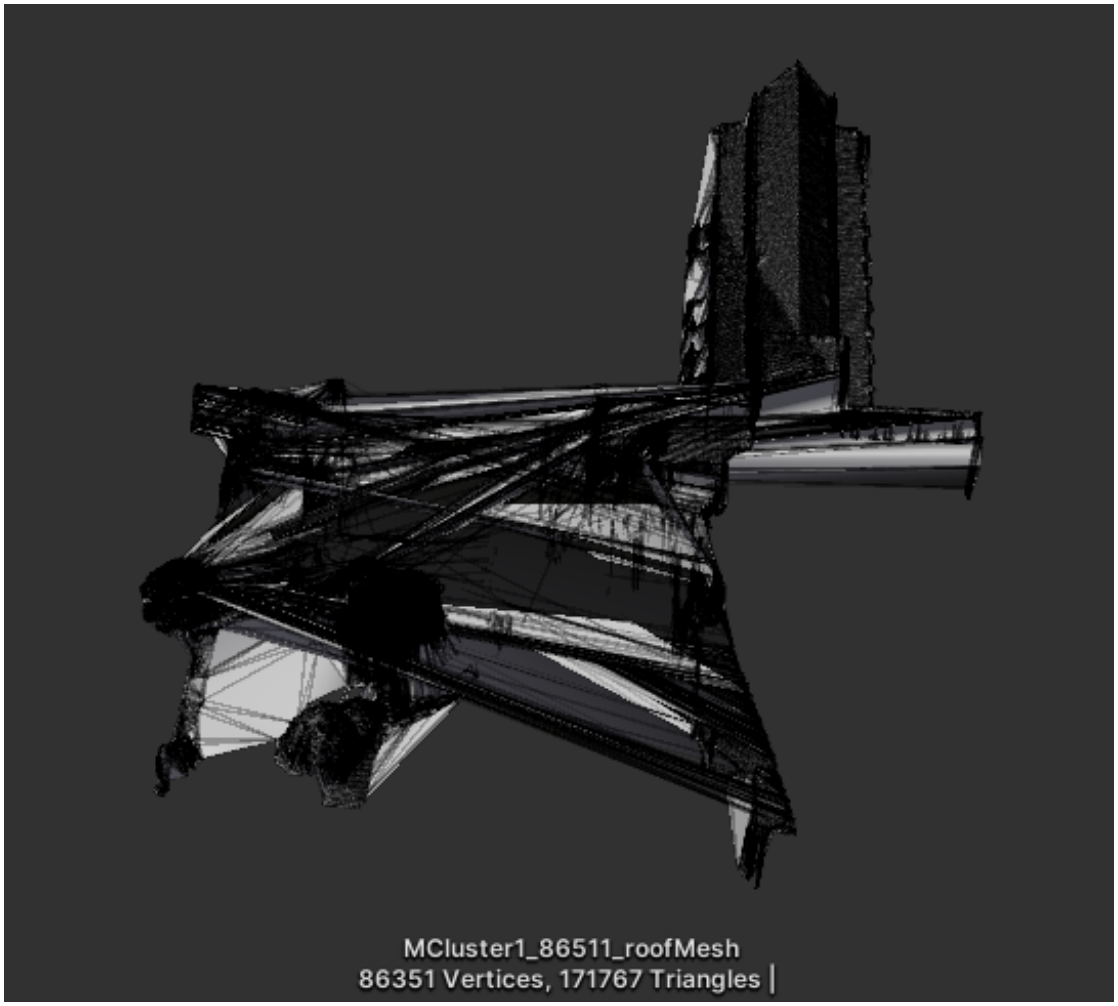

FIGURE D.2: Roof mesh of the original complex showing anomalies: spurious triangles and structural bridges.

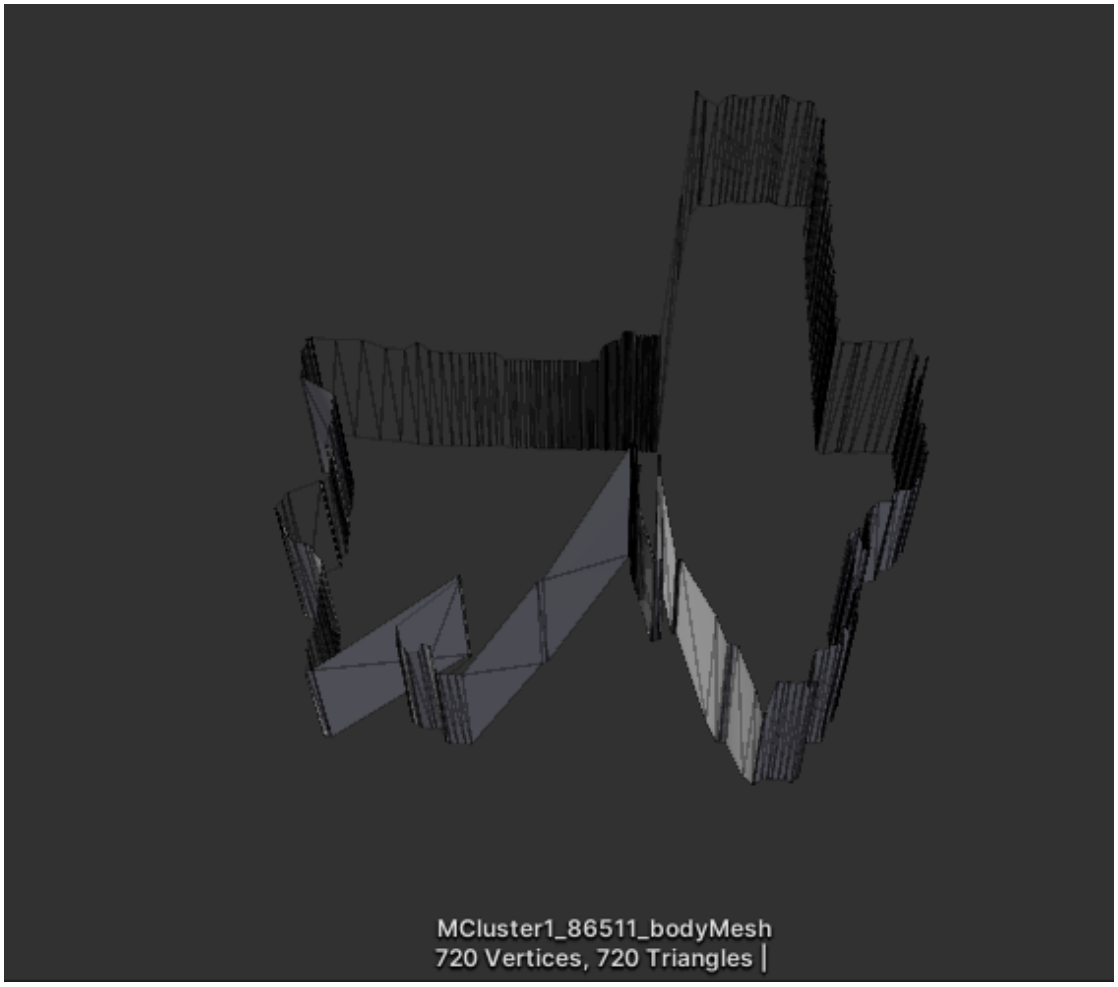

FIGURE D.3: Body mesh of the same complex showing that the boundary estimation was close to correct.

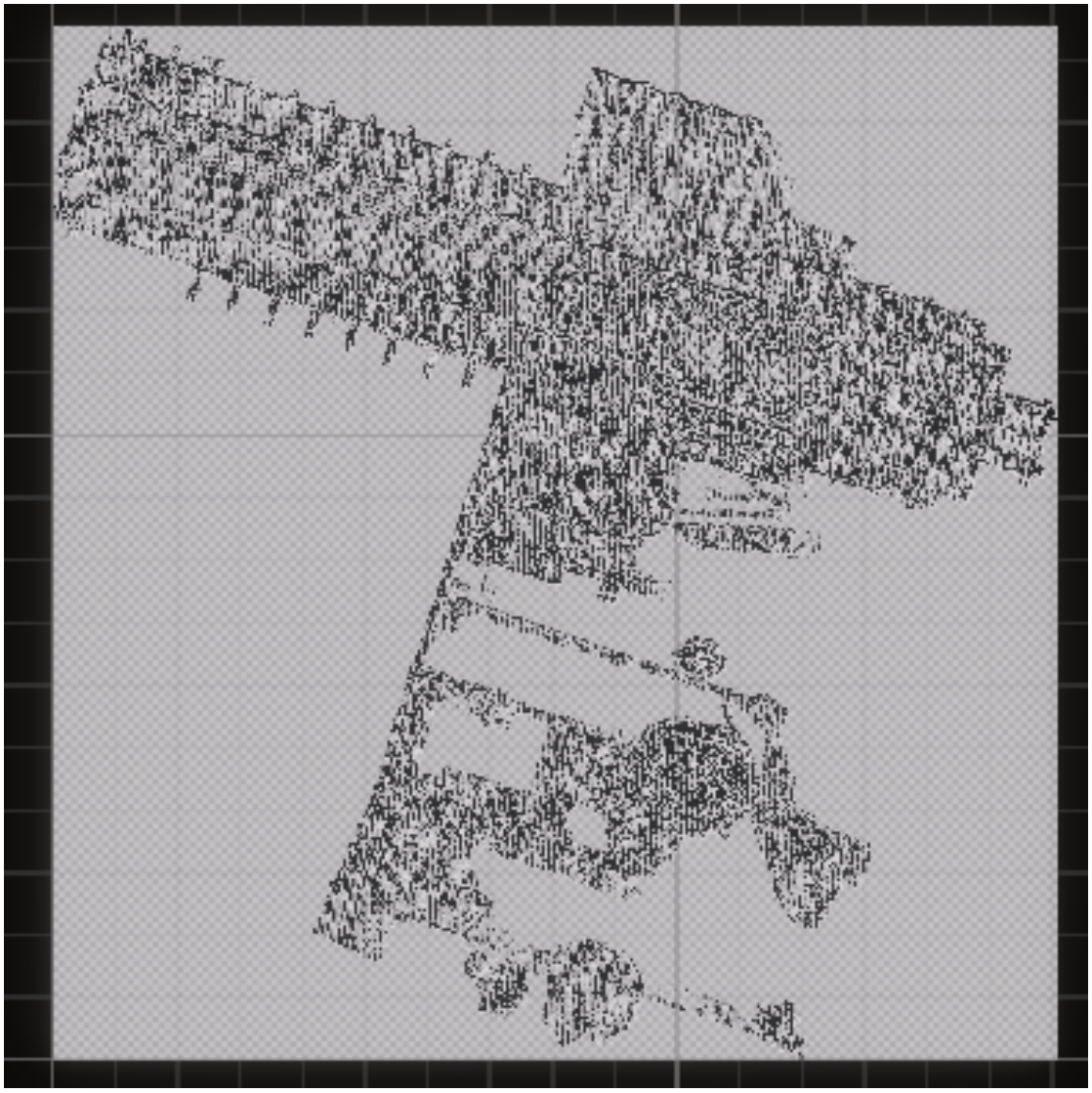

FIGURE D.4: Cathedral complex cluster extracted with a 1.0 m radius threshold (90,544 points). The created collection was used as input for the Terrain Generator. A low resolution and top-down view were used to make the cluster points easier to distinguish.

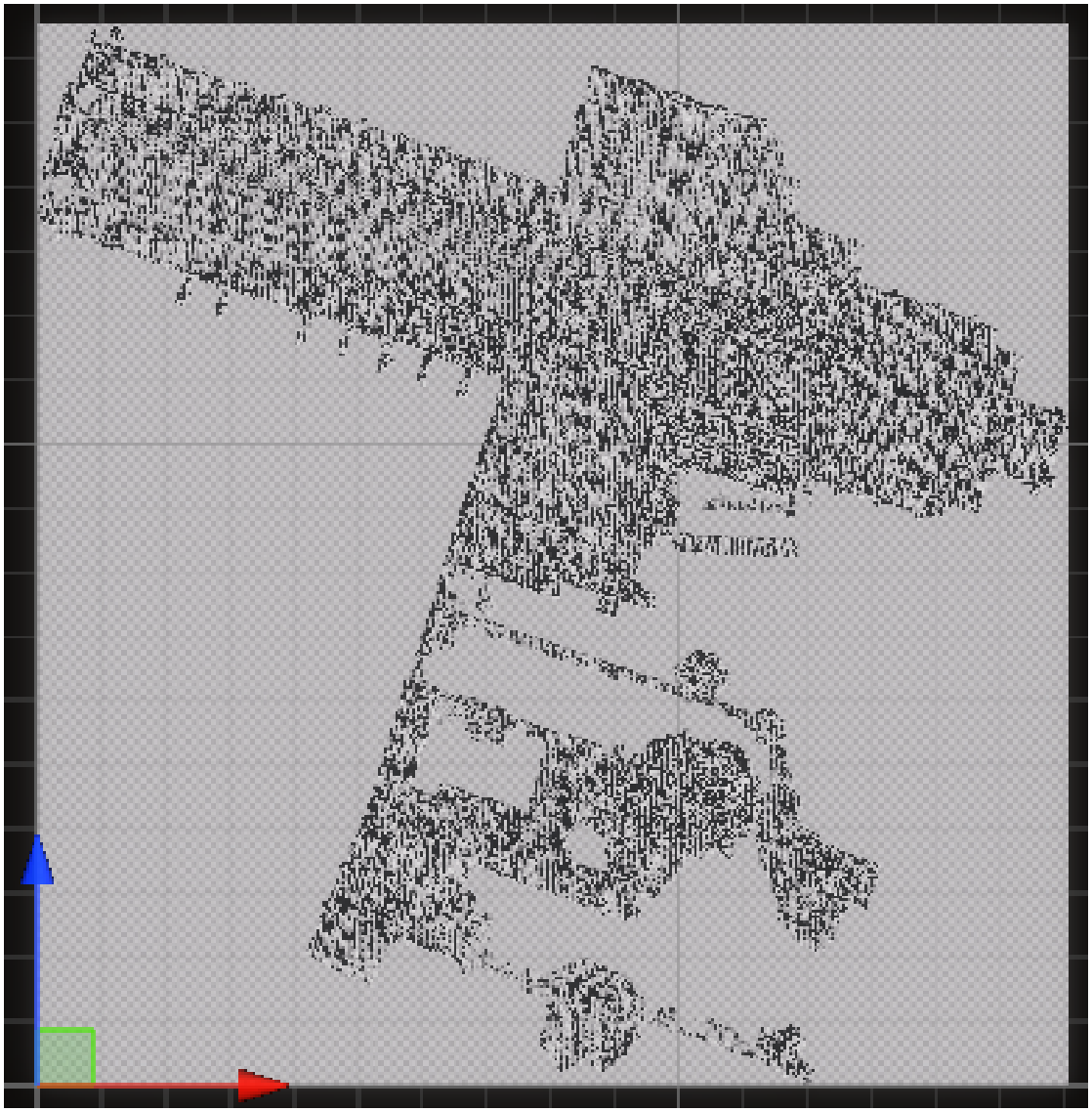

FIGURE D.5: Cathedral complex cluster extracted using an 8.0 m radius threshold. This collection was later used as input for the Terrain Generator. A low resolution and top-down view were used to make the cluster points easier to distinguish. Although subtle, some points were indeed removed — a "spot the difference" challenge.

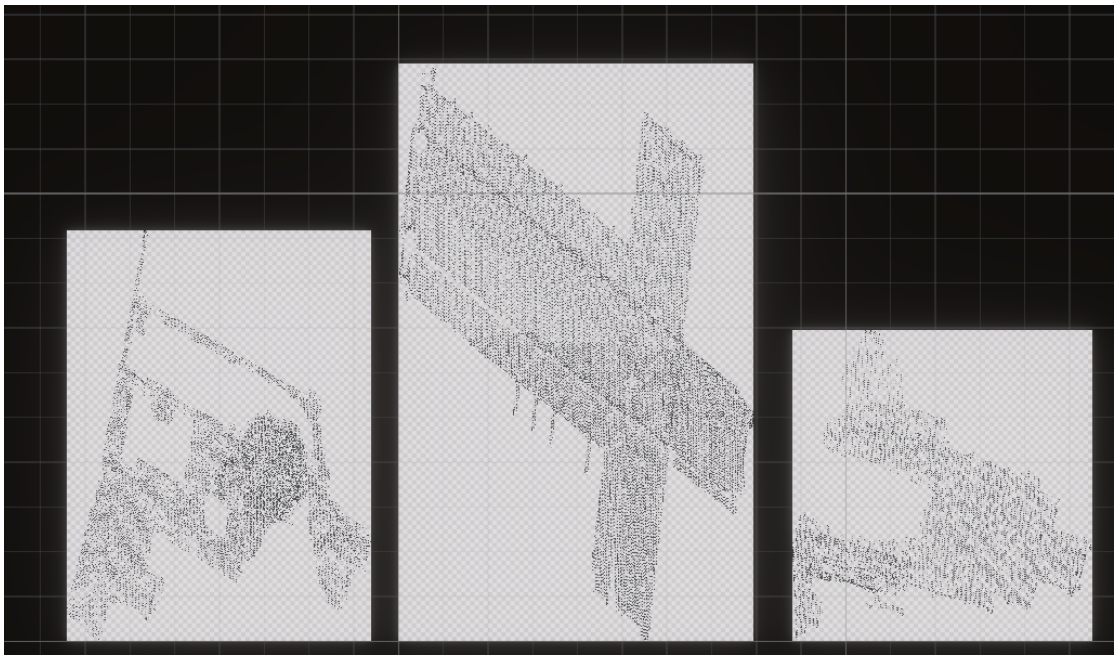

FIGURE D.6: Splitting of the cathedral complex into three subclusters after refining the clustering radius to 0.7 m.

FIGURE D.7: Roof mesh reconstructed from the side building subcluster, with anomalies eliminated.

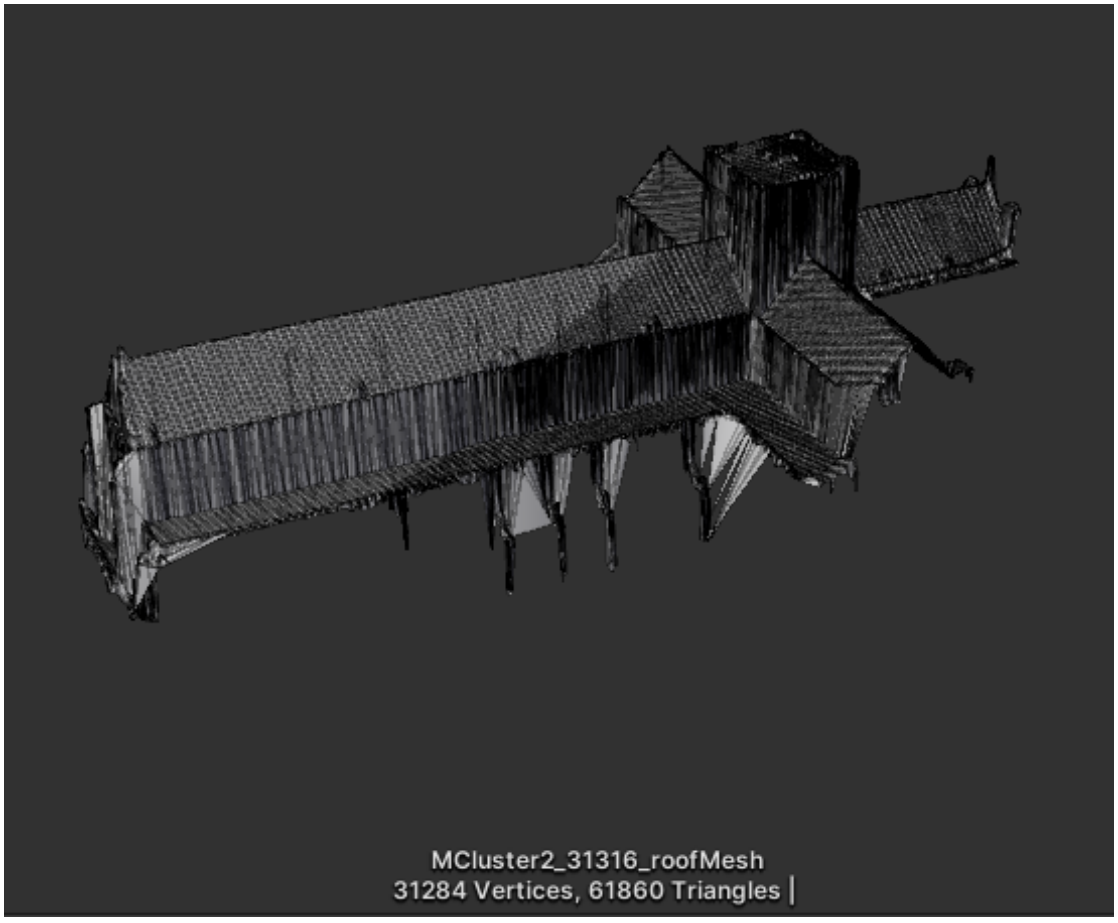

FIGURE D.8: Roof mesh reconstructed from the main nave subcluster, with anomalies eliminated.

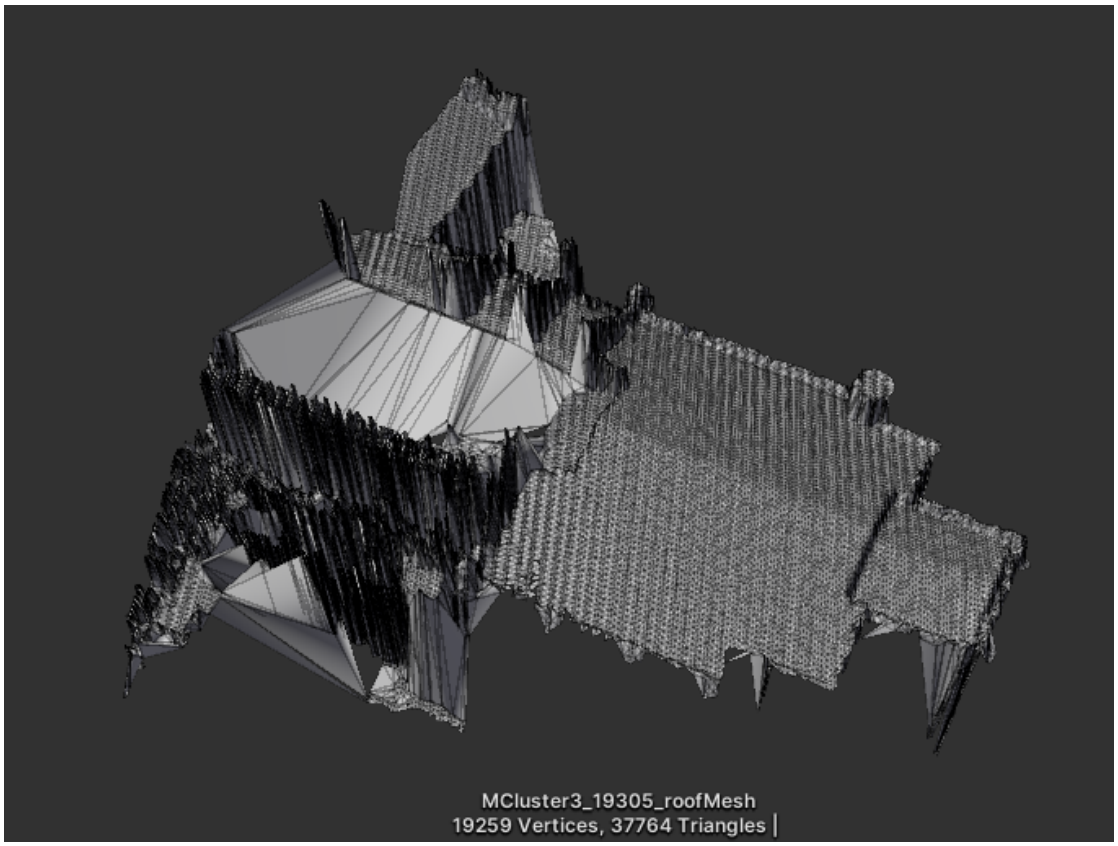

FIGURE D.9: Roof mesh reconstructed from the altar subcluster, with anomalies eliminated.

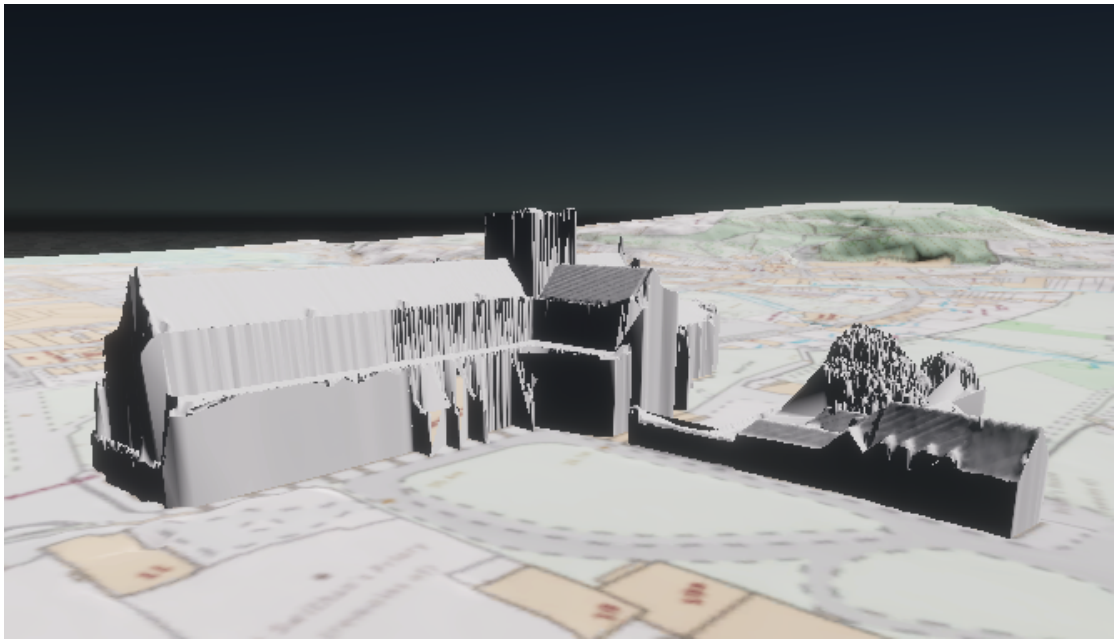

FIGURE D.10: Cathedral complex rebuilt from three subcluster models, accurately placed by the pipeline.

Bibliography

[1] Ziad Abdeldayem. Automatic weighted splines filter (awsf): A new algorithm for extracting terrain measurements from raw lidar point clouds. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 13:60–71, 2020.

[2] Seung Man An. A study on urban-scale building, tree canopy footprint identification and sky view factor analysis with airborne lidar remote sensing data. *Remote Sensing*, 15(15), 2023.

[3] Bartosz Bonczak and Constantine E Kontokosta. Large-scale parameterization of 3d building morphology in complex urban landscapes using aerial lidar and city administrative data. *Computers, Environment and Urban Systems*, 73:126–142, 2019.

[4] Carlos Campoverde, Mila Koeva, Claudio Persello, Konstantin Maslov, Weiqin Jiao, and Dessislava Petrova-Antonova. Automatic building roof plane extraction in urban environments for 3d city modelling using remote sensing data. *Remote Sensing*, 16(8), 2024.

[5] Peter Dorninger and Norbert Pfeifer. A comprehensive automated 3d approach for building extraction, reconstruction, and regularization from airborne laser scanning point clouds. *Sensors*, 8(11):7323–7343, 2008.

[6] EDINA Digimap Ordnance Survey Service. Lidar composite digital surface model england 50cm resolution [asc geospatial data]. Scale 1:2000, Tiles: su4729, su4730, su4829, su4830, 2016. Updated: 5 January 2016, Open Government Licence.

[7] EDINA Digimap Ordnance Survey Service. Lidar composite digital terrain model england 50cm resolution [asc geospatial data]. Scale 1:2000, Tiles: su4729, su4730, su4829, su4830, 2016. Updated: 5 January 2016, Open Government Licence.

[8] EDINA Digimap Ordnance Survey Service. Os mastermap®topography layer [tiff geospatial data]. Scale 1:2000, Tile: su4829, 2016. Updated: 18 May 2016, Ordnance Survey (GB).

[9] Sander Oude Elberink and George Vosselman. Building reconstruction by target based graph matching on incomplete laser data: Analysis and limitations. *Sensors*, 9(8):6101–6118, 2009.

[10] Maolin Feng, Tonggang Zhang, Shichao Li, Guoqing Jin, and Yanjun Xia and. An improved minimum bounding rectangle algorithm for regularized building boundary extraction from aerial lidar point clouds with partial occlusions. *International Journal of Remote Sensing*, 41(1):300–319, 2020.

[11] Erica Marie Gallerani, Lucas Berio Fortini, Christopher C. Warren, and Eben H. Paxton. Identifying conservation introduction sites for endangered birds through the integration of lidar-based habitat suitability models and population viability analyses. *Remote Sensing*, 16(4), 2024.

[12] Getmapping. High resolution (25cm) vertical aerial imagery [jpg geospatial data]. Scale 1:500, Tiles: su4829, 2013. Updated: 11 October 2013, Using: EDINA Aerial Digimap Service.

[13] Sara Gonizzi Barsanti, Marco Raoul Marini, Saverio Giulio Malatesta, and Adriana Rossi. Evaluation of denoising and voxelization algorithms on 3d point clouds. *Remote Sensing*, 16(14), 2024.

[14] Jin Huang, Jantien Stoter, Ravi Peters, and Liangliang Nan. City3d: Large-scale building reconstruction from airborne lidar point clouds. *Remote Sensing*, 14(9), 2022.

[15] Zhenyang Hui, Zhuoxuan Li, Penggen Cheng, Yao Yevenyo Ziggah, and JunLin Fan. Building extraction from airborne lidar data based on multi-constraints graph segmentation. *Remote Sensing*, 13(18), 2021.

[16] Andreas Jochem, Bernhard Höfle, Martin Rutzinger, and Norbert Pfeifer. Automatic roof plane detection and analysis in airborne lidar point clouds for solar potential assessment. *Sensors*, 9(7):5241–5262, 2009.

[17] Karl Kraus and Norbert Pfeifer. Advanced dtm generation from lidar data. *International Archives Of Photogrammetry Remote Sensing And Spatial Information Sciences*, 34(3/W4):23–30, 2001.

[18] Ziming Li, Qinchuan Xin, Ying Sun, and Mengying Cao. A deep learning-based framework for automated extraction of building footprint polygons from very high-resolution aerial imagery. *Remote Sensing*, 13(18), 2021.

[19] Haojie Lian, Kangle Liu, Ruochen Cao, Ziheng Fei, Xin Wen, and Leilei Chen. Integration of 3d gaussian splatting and neural radiance fields in virtual reality fire fighting. *Remote Sensing*, 16(13), 2024.

[20] Bo Liu, Hui Zeng, Qiulei Dong, and Zhanyi Hu. Language-level semantics-conditioned 3d point cloud segmentation. *Remote Sensing*, 16(13), 2024.

[21] Ruzinoor Che Mat, Abdul Rashid Mohammed Shariff, Abdul Nasir Zulkifli, Mohd Shafry Mohd Rahim, and Mohd Hafiz Mahayudin. Using game engine for 3d terrain visualisation of gis data: A review. *IOP Conference Series: Earth and Environmental Science*, 20(1):012037, jun 2014.

[22] Wojciech Matwij, Tomasz Lipecki, and Wojciech Franciszek Jaśkowski. Selection of an algorithm for assessing the verticality of complex slender objects using semi-automatic point cloud analysis. *Remote Sensing*, 16(3), 2024.

[23] Xiaoliang Meng, Tianyi Wang, Dayu Cheng, Wensong Su, Peng Yao, Xiaoli Ma, and Meizhen He. Enhanced point cloud slicing method for volume calculation of large irregular bodies: Validation in open-pit mining. *Remote Sensing*, 15(20), 2023.

[24] Shi Pu and George Vosselman. Building facade reconstruction by fusing terrestrial laser points and images. *Sensors*, 9(6):4525–4542, 2009.

[25] Franz Rottensteiner and Christian Briese. A new method for building extraction in urban areas from high-resolution lidar data. In *International archives of photogrammetry remote sensing and spatial information sciences*, volume 34, pages 295–301. Citeseer, 2002.

[26] Il-Sik Shin, Mohammadamin Beirami, Seok-Je Cho, and Yung-Ho Yu. Development of 3d terrain visualization for navigation simulation using a unity 3d development tool. *Journal of Advanced Marine Engineering and Technology (JAMET)*, 39(5):570–576, 2015.

[27] Zhen Shu, Xiangyun Hu, and Hengming Dai. Progress guidance representation for robust interactive extraction of buildings from remotely sensed images. *Remote Sensing*, 13(24), 2021.

[28] Cristina Soriano-Cuesta, Rocío Romero-Hernández, Emilio J. Mascort-Albea, Martin Kada, Andreas Fuls, and Antonio Jaramillo-Morilla. Evaluation of open geotechnical knowledge in urban environments for 3d modelling of the city of seville (spain). *Remote Sensing*, 16(1), 2024.

[29] Ryan RS Spick and James Walker. Realistic and textured terrain generation using gans. In *Proceedings of the 16th ACM SIGGRAPH European Conference on Visual Media Production*, CVMP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Wouter A. J. Van den Broeck and Toon Goedemé. Combining deep semantic edge and object segmentation for large-scale roof-part polygon extraction from ultrahigh-resolution aerial imagery. *Remote Sensing*, 14(19), 2022.

[31] Jingxue Wang, Dongdong Zang, Jinzheng Yu, and Xiao Xie. Extraction of building roof contours from airborne lidar point clouds based on multidirectional bands. *Remote Sensing*, 16(1), 2024.

[32] Sa Wang, Zhengli Mao, Changhai Zeng, Huili Gong, Shanshan Li, and Beibei Chen. A new method of virtual reality based on unity3d. In *2010 18th International Conference on Geoinformatics*, pages 1–5, 2010.

[33] Yong Wang, Xiangqiang Zeng, Xiaohan Liao, and Dafang Zhuang. B-fgc-net: A building extraction network from high resolution remote sensing imagery. *Remote Sensing*, 14(2), 2022.

[34] Liegang Xia, Junxia Zhang, Xiongbo Zhang, Haiping Yang, and Meixia Xu. Precise extraction of buildings from high-resolution remote-sensing images based on semantic edges and segmentation. *Remote Sensing*, 13(16), 2021.

[35] Dong Yang, Jingyuan Wang, and Xi Yang. 3d point cloud shape generation with collaborative learning of generative adversarial network and autoencoder. *Remote Sensing*, 16(10), 2024.

[36] Shuai Zhang, Manchun Li, Zhenjie Chen, Tao Huang, Sumin Li, Wenbo Li, and Yun Chen. Parallel spatial-data conversion engine: Enabling fast sharing of massive geospatial data. *Symmetry*, 12(4), 2020.

[37] Pengbo Zhou, Li An, Yong Wang, and Guohua Geng. Mlgtm: Multi-scale local geometric transformer-mamba application in terracotta warriors point cloud classification. *Remote Sensing*, 16(16), 2024.

[38] Martin Štroner, Rudolf Urban, and Lenka Línková. Color-based point cloud classification using a novel gaussian mixed modeling-based approach versus a deep neural network. *Remote Sensing*, 16(1), 2024.